TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky a mezioborových studií

Studijní program: N2612 – Elektrotechnika a informatika
Studijní obor: 1802T007 – Informační technologie

Rekonstrukce textur pro 3D model získaný z 2D dat

Texture reconstruction for 3D model obtained from 2D data

Diplomová práce

Autor: **Bc. Petr Ječmen**
Vedoucí práce: Ing. Petr Kretschmer

V Liberci 20. 5. 2011

Místo pro vložení zadání

Prohlášení

Byl(a) jsem seznámen(a) s tím, že na mou diplomovou práci se plně vztahuje zákon č. 121/2000 o právu autorském, zejména § 60 (školní dílo).

Beru na vědomí, že TUL má právo na uzavření licenční smlouvy o užití mé diplomové práce a prohlašuji, že **s o u h l a s í m** s případným užitím mé diplomové práce (prodej, zapůjčení apod.).

Jsem si vědom(a) toho, že užít své diplomové práce či poskytnout licenci k jejímu využití mohu jen se souhlasem TUL, která má právo ode mne požadovat přiměřený příspěvek na úhradu nákladů, vynaložených univerzitou na vytvoření díla (až do jejich skutečné výše).

Diplomovou práci jsem vypracoval(a) samostatně s použitím uvedené literatury a na základě konzultací s vedoucím diplomové práce a konzultantem.

Datum

Podpis

Poděkování

Hlavní poděkování patří mé rodině, která mě po celou dobu studia podporovala a vytvářela co nejlepší podmínky, abych mohl zdárně dokončit studium a vypracovat diplomovou práce dle vlastních představ.

Další velké díky patří vedoucímu mé práce, Ing. Petru Kretschmerovi. Ten jednak téma vymyslel, navíc mi po celou dobu pomáhal radami a konzultacemi, které mi umožnili si utřídit myšlenky a nápady na správná místa, takže jsem mohl pokračovat v práci bez větších zádrhelů.

Abstrakt

Tato práce se zabývá problematikou rekonstrukce textury prostorového modelu z fotografií. V první části se nachází teoretický rozbor problému, identifikace dílčích úloh algoritmu a jejich detailní teoretický rozbor. Některé úlohy šlo řešit více způsoby, kapitoly pak obsahují popis všech možných postupů společně se zhodnocením, který postup je nejvhodnější a z jakého důvodu. Druhá část se zabývá praktickou implementací v jazyku Java a zakomponováním do již existující aplikace „Extrakce 3D grafického modelu z 2D dat.“ Nejedná se o vyčerpávající popis kompletního zdrojového kódu, tato část se spíše soustředí na zajímavé či důležité části kódu. Zhodnocením kvality rekonstrukce a její rychlostí se pak zabývá poslední část práce. V té se také nacházejí návrhy na možná budoucí rozšíření a vylepšení aplikace pro rekonstrukci modelu objektu.

Klíčová slova: rekonstrukce, textura, model, 3D

Abstract

This thesis focuses on texture reconstruction of object model from photos. First path contains theoretical summary of given problem, main tasks of algorithm are identified and described in detail. Some of the tasks have more than one possible approach to the solution, so chapters with those tasks contain description of all possible approaches and discussion which approach has been chosen and why. Second part is about concrete implementation in programming language Java and incorporation into existing application “3D graphic model extraction from 2D data.” This part is not exhaustive description of whole source code; it is rather description of important or interesting parts of algorithm. Quality and speed of reconstruction is discussed in last part of the thesis. It also contains possible extensions and improvements for program for object model reconstruction.

Keywords: reconstruction, texture, model, 3D

Obsah

Úvod

Počítačové modelování je v dnešní době velice rozvinutý obor. Využití nalézá v prakticky každém myslitelném oboru. Pokud chceme něco vyrobit, nejjednodušší postup je právě vytvoření počítačového modelu., který předáme výrobci a on podle něj daný kus vyrobí.

Modelování se ale netýká pouze výroby předmětů, další velkou oblastí, kde se modelování používá, je počítačová grafika. Moderní filmy spoléhají na velice složité prostorové modely, které jsou zasazovány do scény a jsou k nerozeznání od reálných předmětů. Podobné modely se také používají v počítačových hrách, které na modelech prakticky stojí.

Fenoménem posledních let jsou interaktivní mapy Země, které kromě „obyčejného“ pohledu na tvary země umožní sledovat místa ideálně ve třech rozměrech. Prvním krokem k realizaci této myšlenky bylo pořízení leteckých fotografií všech míst, které po nanesení na výškovou mapu povrchu vytvoří vcelku reálnou krajinu. Toto je ale vhodné pouze pro velké objekty, či spíše pro samotnou krajinu, protože výšková mapa Země jsou data o terénu, nikoliv o objektech na ní.

Získání informací o objektech na povrchu krajiny (hlavním cílem jsou budovy) je proces velice komplexní, který není jednoduché zautomatizovat. Vývoj jde v této oblasti rychle kupředu, přesto ale zatím není k dispozici software, který by dokázal hromadně zpracovávat fotografická data a tvořit z nich modely. Momentálně se vývoj hlavně zaměřuje na rekonstrukci modelů jednotlivých objektů, ideálně bez asistence uživatele.

Tato práce se takovouto rekonstrukcí zabývá, konkrétně se zaměřuje na rekonstrukci textur objektu. Pro rekonstrukci drátěného modelu objektu je použit software vytvořený v rámci bakalářské práce [Ječmen2009]. Používá se jeho portovaná verze do jazyka Java, která vznikla v rámci diplomového projektu [Ječmen2010].

Drátěný model může být v mnoha aplikacích dostačující (typicky výroba objektu), někdy je ale vhodné či dokonce nezbytné znát i vzhled objektu. Jednou z hlavních aplikací jsou výše uvedené 3D mapy, kde pouhý tvar není příliš vypovídající. Proto bylo přistoupeno k rozšíření aplikace „Extrakce 3D grafického modelu z 2D dat“ o část, která by realizovala rekonstrukci textur daného objektu a samozřejmě by pak byla schopná výsledky nějak prezentovat a uložit.

1. Teoretický rozbor

V této kapitole se jednak seznámíme s procesem rekonstrukce jako celku – co je nutné k výpočtu, z jakých částí se výpočet skládá a také co se očekává jako výstup rekonstrukce.

Fotoaparát k zaznamenání používá, podobně jako lidské oko, perspektivní projekci. Ta umožňuje transformovat body z prostoru na body nacházející se na 2D ploše. Matice (**1)** představuje matici perspektivního zobrazení. Parametr *d* určuje vzdálenost plochy, na kterou se body budou zobrazovat, často se také nazývá ohnisková vzdálenost (lze se setkat s označením f).

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{d} & 0 \end{bmatrix} \qquad \textbf{(1)}$$

Důležitou vlastností této matice je fakt, že pro jeden bod na 2D ploše existuje nekonečně mnoho bodů v prostoru, které lze transformovat na bod na ploše. Tato skutečnost komplikuje proces rekonstrukce, protože nelze přímo promítnout bod zpět z fotografie do prostoru za pomoci inverzní matice. Byli bychom schopní získat rovnici polopřímky, na které se daný bod fotografie může v prostoru nacházet, čehož se využívá při rekonstrukci polohy rohových bodů modelu v prostoru, pro rekonstrukci textury se ale používá jiný přístup.

Rekonstrukce se děje pomocí zpětné projekce bodů z prostoru na fotografii. Před vlastním promítáním je však nutné vykonat několik kroků, které připraví parametry nutné pro výpočet. V teoretickém rozboru bude, jak název napovídá, rozebrán analytický popis postupu, implementaci je pak věnována samostatná část.

1.1. Analýza algoritmu rekonstrukce

Postup použitý v této práci je přesným opakem projekce bodů z fotografie do prostoru. Nejdříve nalezneme polohu bodů v prostoru a ty pak promítneme zpět na 2D plochu fotografie. Před vlastním promítáním je ale nutné nalézt nejvhodnější fotografii, na kterou budeme dané body promítat. Co určuje nejvhodnější fotografii blíže popíšeme v kapitole **1.2**. Je nutné zavést jednotný popis kvality fotografie, který bude schopen ohodnotit libovolnou fotografii dle její vhodnosti pro výpočet. I když vlastní polygon je v našem případě trojúhelník, grafická data se ukládají do obecně obdélníkového rastru, musíme tedy náš trojúhelník obalit do obdélníku (viz. kapitola **1.3**). Obalení do obdélníku nám navíc usnadní výpočet bodů ležících uvnitř trojúhelníku.

Prozatím se stále nacházíme ve spojitém prostoru $\mathbf{R}^3$, pro výpočet je ale nutné, aby počet počítaných bodů byl konečný. Musíme tedy požadovanou oblast pokrýt sítí bodů, na kterých bude probíhat výpočet (kapitola **1.4**). Probereme možné metody pokrytí spojité oblasti sítí bodů, z kterých vybereme jednu, kterou rozebereme podrobněji a také ji použijeme ve výpočtu. Předposledním (a hlavním) krokem algoritmu je zpětná projekce bodů z prostoru na plochu fotografie pomocí perspektivní projekce. Této části se věnuje kapitola **1.5**. Kromě vlastní projekce na plochu fotografie se budeme věnovat metodám interpolace, protože získaná poloha na fotografii není obecně celočíselná. Posledním krokem je pak výpočet souřadnic textury. Tento krok je nutný, protože trojúhelník může být do obdélníku textury vložen téměř libovolně, je tedy nutné uchovávat souřadnice třech bodů tvořících náš trojúhelník (viz. kapitola **1.6**).

1.2.Nalezení nejvhodnější fotografie

Pro vyhodnocení kvality fotografie z hlediska rekonstrukce je nutné zvolit konečný počet parametrů, pomocí kterých bude možno jednoznačně určit vhodnost dané fotografie. Výpočet by měl být jednak rychlý, měl by ale také být stabilní, čili jestliže spustíme výpočet opakovaně pro stejnou fotografii, měli by nám vyjít stejné výsledky.

Jako primární parametr byla zvolena plocha rekonstruovaného trojúhelníku na dané fotografii. Plocha byla zvolena hlavně kvůli konečné interpolaci barev výstupní textury. Je velice pravděpodobné, že výstupní textura bude větší než trojúhelník na fotografii, budeme tedy nuceni dopočítávat barvu v některých bodech textury. Toto dopočítávání ale není nikdy přesné, protože neznáme funkci charakterizující barvu našeho objektu. Známe pouze barvy v konečném počtu diskrétních bodů, hodnoty mezi nimi musíme dopočítat pomocí metod interpolace (podrobnosti v kapitole **1.5**).

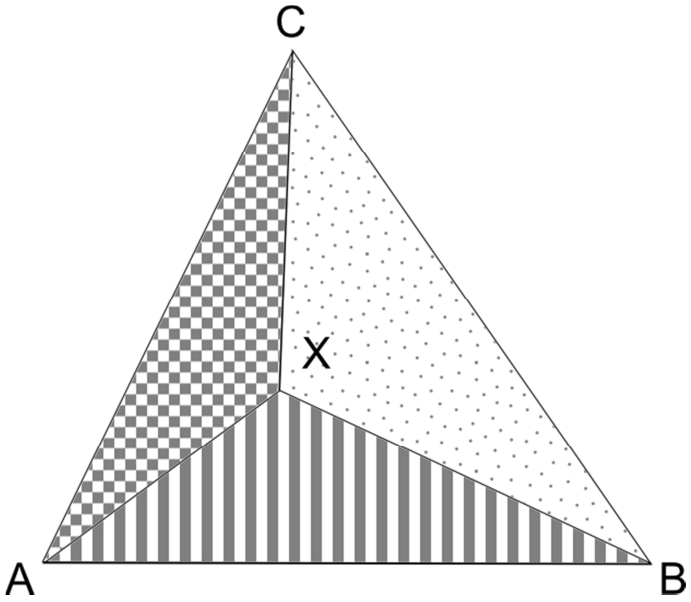
Jako další vhodný parametr by se mohlo zdát natočení / otočení kamery vůči trojúhelníku. Ta charakterizuje, jak bude trojúhelník na fotografii deformován – ideální případ je, že zobrazovací rovina fotoaparátu je rovnoběžná s plochou trojúhelníku, trojúhelník je pak na fotografii zobrazen přímo jako použitelná textura. V běžném případě se ale natočení pohybuje kolem 45°, textura tedy bude nějak deformována a budeme ji muset narovnávat. Nalezení hodnotící funkce, která by vhodně charakterizovala vliv rotace na deformaci textury, bylo vyhodnoceno jako příliš složité, při dostatečné ploše textury lze za použití vhodného matematického aparátu texturu narovnat bez větších chyb i při velkých úhlech. Proto byl tento parametr zavrhnut.

Kromě geometrických vlastností trojúhelníku se nabízí možnost zjistit, jak moc se mění na fotografii osvětlení či jestli se na ní nenacházejí nežádoucí odlesky. Řešením této úlohy se zabývají algoritmy počítačového vidění, bohužel ale vyžadují hluboké znalosti v tomto oboru. K dostatečnému prostudování této problematiky nebyl dostatek času, tento parametr byl tedy bohužel vynechán. Jedná se ale o velice účinný ukazatel nežádoucího poškození textury, bylo by tedy vhodné v rámci dalšího vylepšování aplikace tuto část zdůraznit.

Toto byl přehled parametrů (parametru), které tvoří základní skóre fotografie. Je nutné ale zavést také systém penalizací, který zajistí, že se při výpočtu nepoužijí fotografie, na kterých je textura nějak zakrytá, či dané fotografie alespoň jasně označí.

První penalizační faktor je detekce překrytí trojúhelníku jiným objektem (částí modelu). Vzhledem k tomu, že není dost dobře možné získat sémantický popis scény

(co která plocha reprezentuje, je-li něco překryto nebo zakryto), musíme pro detekci zakrytí použít data poskytnutá uživatelem. K výpočtu nám stačí seznam bodů, které se nacházejí na fotografii. Z nich vybereme tři body, které tvoří náš trojúhelník, a otestujeme polohu ostatních bodů vůči našemu trojúhelníku. Polohy mohou být dvě – buď leží uvnitř trojúhelníku, nebo ne. K vyhodnocení polohy je použita metoda porovnání obsahu trojúhelníků.



Obrázek 1 Bod X leží uvnitř trojúhelníku

Obrázek 1 znázorňuje případ, kdy bod X leží uvnitř trojúhelníku. Můžeme si všimnout, že součet obsahů trojúhelníků ABX, BCX a ACX je rovna obsahu trojúhelníku ABC.

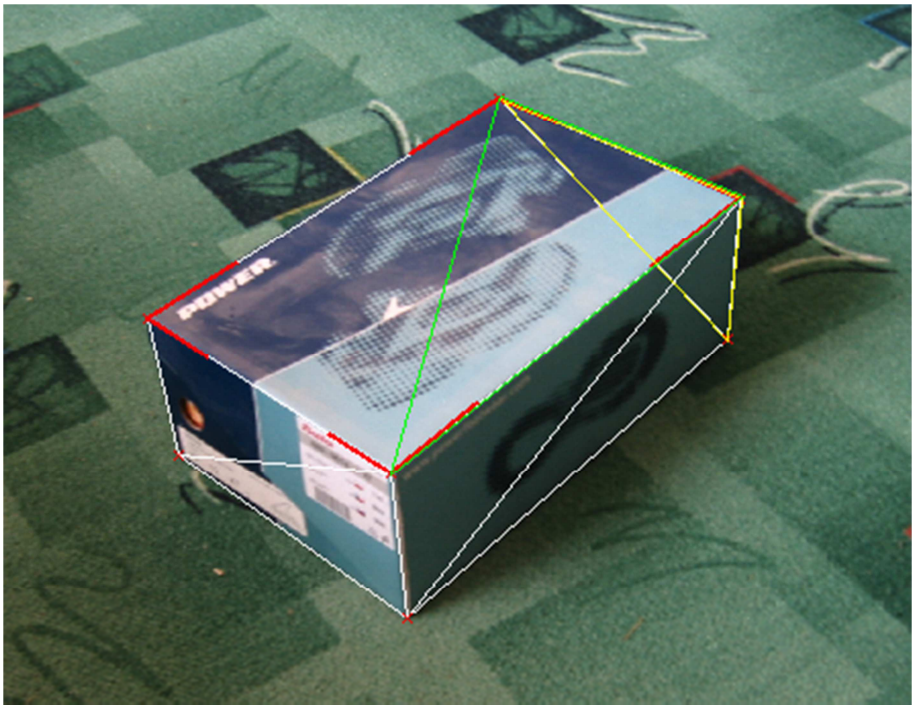
Pokud by bod X neležel uvnitř trojúhelníku, součet obsahů trojúhelníků by pak byl různý od obsahu našeho velkého trojúhelníku. Pro výpočet obsahu trojúhelníku byla použita Heronova formule **(2)** (a je délka strany AB, b pro stranu BC a c pro stranu AC).

$$S = (s(s-a)(s-b)(s-c))^{\frac{1}{2}}, \; s = \frac{a+b+c}{2} \qquad \textbf{(2)}$$

Odvození a platnost vzorce lze nalézt v prakticky libovolné literatuře týkající základů geometrie (např. [Švrček1988]). Tento vzorce byl zvolen z hlediska vstupních údajů, kdy máme k dispozici pouze souřadnice bodů, nikoliv výšky v trojúhelnících, které bychom museli dopočítávat.

Ve finále tedy získáme dvě čísla – součet obsahu „menších“ trojúhelníků a obsah hlavního trojúhelníku. Pokud se tato čísla liší, bod leží mimo trojúhelník a tím pádem ho nezakrývá. V opačném případě je nutné označit danou fotografii jako nevhodnou, protože je textura zakrytá jiným objektem / jinou částí modelu.

Druhá metoda penalizace není již pouze rozhodnutí, jestli se daná fotografie dá použít či ne. Jedná se také o detekci zakrytí textury, detekce ale již není stoprocentní a může zaznamenat mnoho falešných detekcí. Přesto je ale vhodné tuto penalizaci použít, je ale důležité vhodně zvolit koeficient penalizace. Princip detekce zakrytí této metody je založen na detekci průsečíku přímek. Pokud některou se stran našeho trojúhelníku prochází nějaká jiná úsečka, je pravděpodobné, že je textura zakrytá (není to ale nutně pravda).



Obrázek 2 Překrytí trojúhelníku jiným

Obrázek 2 ukazuje typický případ překrytí trojúhelníků. Zelený trojúhelník tvoří část vrchní strany předmětu a z fotografie je patrné, že je plně viditelný. Žlutý trojúhelník tvoří část boční strany a na této fotografii není z jeho povrchu vidět ani část. Jak je patrné, žlutá a zelená křivka se protínají, dojde tedy k detekci překrytí. Z hlediska zeleného trojúhelníku se jedná o detekci falešnou, u žlutého je to detekce oprávněná.

Toto chování má jednoduchý důvod. Celý model je tvořen trojúhelníky. Při určitých pohledech na model je šance, že budou vidět body tvořící trojúhelník, přitom vlastní plocha trojúhelníku bude schována (odvrácena). Na obrázku 2 je tento případ vidět. Body žlutého trojúhelníku jsou na fotografii vidět, díváme se přitom na „odvrácenou“ stranu trojúhelníku, která je navíc překryta horní stěnou. Vzhledem k nejistotě v detekci těchto zakrytí bylo přistoupeno k řešení, ve kterém se sečte počet těchto možných zakrytí a dle této hodnoty se upraví výsledné skóre fotografie (typicky se vydělí nějakým číslem získaným z počtu detekcí).

Poslední penalizací je kvalita rekonstrukce polohy a orientace kamery příslušné fotografie. Toto číslo si drží výpočetní engine z doby, kdy byl rekonstruován drátěný model objektu, stačí tedy pouze toto číslo načíst a použít. Číslo nebudeme dále nijak upravovat, pouze jeho hodnotu zahrneme do penalizačního ukazatele fotografie

Výstupem těchto kroků je číslo, které charakterizuje vhodnost dané fotografie pro rekonstrukci textury. Tento postup zopakujeme pro všechny fotografie a vybereme tu s největším skóre. Vzhledem k možnosti falešných detekcí překrytí bylo rozhodnuto, že bude možné, aby uživatel ručně vybral nejvhodnější fotografii. Mohou nastat případy, kdy fotografie je velice kvalitní (z hlediska rekonstrukce), falešné detekce zákrytů ale mohou degradovat výsledné skóre natolik, že fotografie nebude vybrána.

1.3.Příprava omezujícího obdélníku

Po vybrání nejvhodnější fotografie můžeme přistoupit k přípravám zpětné projekce bodů z prostoru na fotografii. Tento krok by se teoreticky mohl provést již před výběrem nejvhodnější fotografie, protože používá data finálního modelu, byl ale navržen postup, který využije informací o bodech na fotografii pro výpočet polohy trojúhelníku v prostoru. Ten by měl snížit vliv chyby rekonstrukce drátěného modelu na přesnost rekonstruované textury.

V této práci budou zmíněny oba postupu, jak ten, který používá data výsledného modelu, tak ten, který používá informace z fotografie. Postupy se liší v tom, jak se získává pozice bodů našeho trojúhelníku v prostoru, postup jak se trojúhelník obalí obdélníkem je pak shodný pro oba přístupy.

První postup pouze vezme pozice bodů z dat výsledného drátěného modelu a obalí je do obdélníku (postup obalení bude zmíněn na konci této kapitoly, protože je shodný pro oba postupy).

Druhý postup napodobuje přístup použitý při rekonstrukci pozic bodu modelů v prostoru. Kromě vlastních fotografií potřebuje ještě data o kamerách (pozice, natočení a vnitřní parametry). Poloha bodů v prostoru se počítá jako průsečík dvou a více (počet záleží na počtu dostupných fotografií) polopřímek v prostoru. Obecně se jedná o mimoběžky, hledá se tedy střed jejich příčky (nejkratší úsečka, která spojuje obě polopřímky). Detailní popis této metody lze nalézt v [Ječmen2009]. Při tomto postupu dochází k odchýlení výsledného bodu od všech polopřímek (bod prakticky nikdy ve výsledku neleží ani na jedné z použitých polopřímek), dochází tedy k pozměnění geometrie výsledného modelu vůči geometrii zachycené na fotografii. Z pohledu rekonstrukce drátěného modelu má tento postup své opodstatnění, pro potřeby rekonstrukce textur je tento jev nežádoucí.

Byl tedy navržen postup, který toto chování eliminuje (i za cenu nepřesné polohy bodu v prostoru, pro nás je důležité zachování geometrie). Jak bylo zmíněno výše, pro výpočet polohy bodu se používá střední bod příčky. Pro naše potřeby se algoritmus modifikoval tak, že se jako výsledný bod bral krajní bod příčky, který leží na polopřímce získané z naší vybrané fotografie. Tento postup se zopakuje pro všechny dostupné (a vhodné) fotografie. Tímto postupem získáme sadu bodů ležících na nosné polopřímce, ty pouze zprůměrujeme a získáme výslednou polohu jednoho z bodů trojúhelníku v prostoru. To zopakujeme i pro ostatní dva body trojúhelníku.

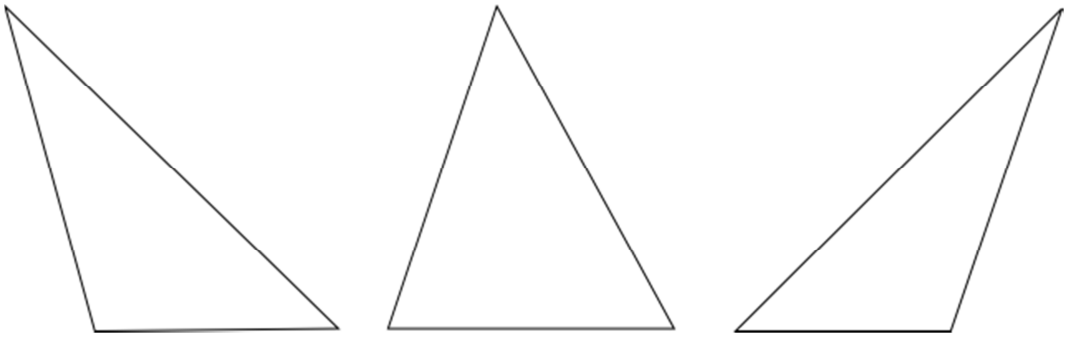
V tomto momentě máme tedy k dispozici polohu tří bodů v prostoru, které tvoří náš trojúhelník. Výsledná textura je ale uložena ve čtverci, potřebujeme tedy alespoň obdélník, abychom byli schopni dobře umístit data do textury. Základním krokem je vhodné zvolení souřadné soustavy, to znamená vhodně zvolit osy, kterých se budeme držet. Osa X byla zvolena ve směru spojnice prvních dvou bodů, osa Y je pak kolmice na ni ve směru třetího bodu (jsme ve 3D, kolmice na přímku není tedy definována jednoznačně, je nutné ji omezit směr nějakým bodem). Pro potřeby výpočtu budeme body trojúhelníku označovat A, B, C. Vektory budou psány jako dvojice tučných písmen – **AB**, které označují, odkud a kam vektor vede. Výpočet takovéhoto vektoru se pak realizuje odečtením cílového bodu od počátečního (B - A).

Osa X se vypočte jako normalizovaný vektor **AB**. Získat osu Y není tak přímočaré, opět ale stačí základní znalosti analytické geometrie. Postup by byl slovně popsán zhruba takto – počáteční bod směrového vektoru osy Y je tvořen projekcí bodu C na osu X. Koncový bod směrového vektoru je pak samotný bod C. Takto vypočtený vektor stačí normalizovat a máme k dispozici vektor pro osu Y. Matematický zápis výše uvedeného postupu je vidět ve vzorci **(3)**.

$$\boldsymbol{Y} = \frac{C - C_X}{|C - C_X|} \qquad \textbf{(3)}$$

$$\boldsymbol{C_X} = (X * (\boldsymbol{AC} * X)) + A$$

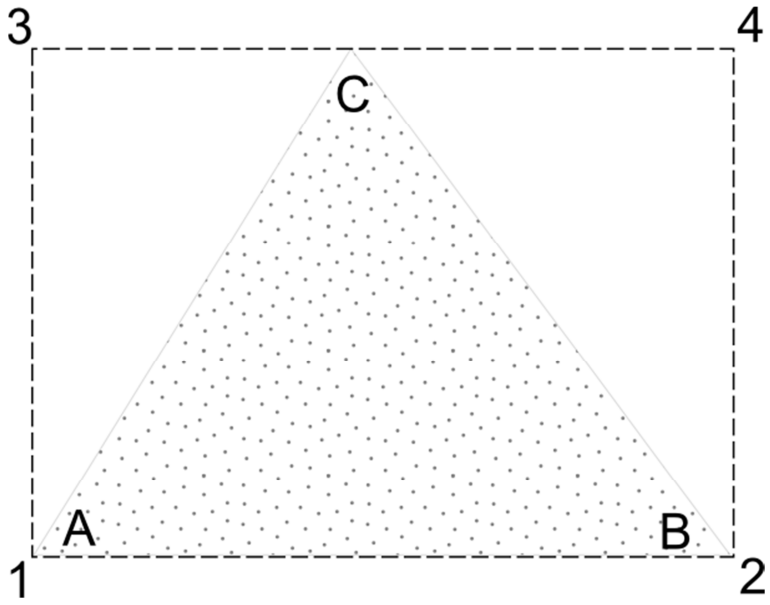
Nyní máme k dispozici jak body trojúhelníku, tak osy X a Y. Bohužel ještě nemůžeme přímo přistoupit k výpočtu bodů omezujícího trojúhelníku. Mohou totiž nastat tři případy polohy bodu C v naší nově vypočítané souřadné soustavě.



Obrázek 3 Možné polohy bodu C (vrchního) vůči základně

Obrázek 3 tyto situace ukazuje – bod C (vrchní) se nachází buď nalevo od základny, nad ní či napravo od ní. Pro zjištění polohy bodu C lze použít projekci bodu C_X, či spíše vektoru jdoucího z A do C_X. Pokud směřuje opačným směrem než **AB**, pak C je nalevo od základny, v opačném případě je napravo. Pokud se bod C nachází nad základnou, pak délka $\mathbf{AC_X}$ bude menší než délka **AB**. Pomocí těchto pravidel zjistíme polohu C a vypočteme příslušné pozice omezujícího trojúhelníku. Přesný postup výpočtu bude zmíněn až v implementační části, jedná se pouze o správnou manipulaci se souřadnicemi bodů A, B, C a jejich přiřazování bodům omezujícího obdélníku.

Obrázek 4 znázorňuje případ, kdy se bod C nachází nad základnou a trojúhelník je obalen do obdélníku. Čísla u bodů obdélníku znázorňují, jak jsou body uloženy. Postup proti hodinovým ručičkám nemá nějaký hlubší důvod, byl zvolen z hlediska postupu iterace např. v cyklu (zleva doprava, odspoda nahoru), navíc body 1, 2 a 3 tvoří osy X a Y.



Obrázek 4 Výsledné obalení trojúhelníku

1.4.Rasterizace obdélníku

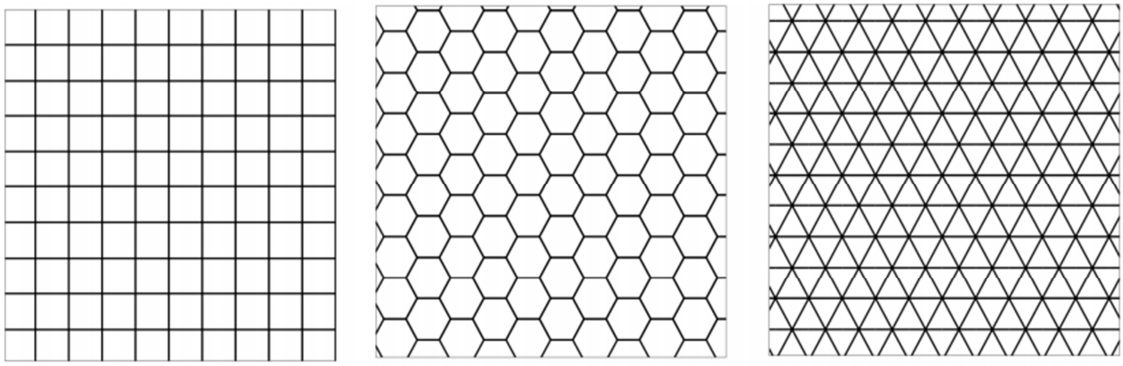
Obalení trojúhelníku se může zdát jako nepříliš významný krok, nyní se ale přesvědčíme, že zavedením unifikovaného souřadného systému (vedlejšího efektu obalení) si velice usnadníme další kroky výpočtu.

Doposud jsme veškeré operace prováděli ve spojité oblasti (konkrétně v prostoru $\mathbf{R}^3$), vlastní rekonstrukce ale musí proběhnout v diskrétní oblasti (výsledek je 2D textura tvořená konečným počtem bodů). Musíme tedy prostor obdélníku rozdělit na konečný počet podoblastí, jejichž krajní body budou tvořit množinu bodů, které se budou projektovat zpět na fotografii.

Prvním krokem v rozdělení prostoru na menší oblasti je volba, zda dělení bude pravidelné, či zvolíme nějaké kritérium, kterým se bude určovat hustota a velikost jednotlivých podoblastí. Nepravidelná struktura se typicky používá pro matematicko-fyzikální výpočty, ve kterých se úlohy skládají z výpočtu rovnic pro ohromné množství oblastí. Snahou je tedy minimalizace počtu těchto oblastí při zachování kvality výpočtu. Existují tedy algoritmy, které dokážou identifikovat oblasti, ve kterých není důležité velké množství podoblastí (typicky velké plochy) a také oblasti, kde by bylo vhodné větší množství menších podoblastí, protože v tom místě dochází k velkým změnám počítané veličiny.

V oblasti zpracování obrazu se ale tento přístup prakticky nepoužívá. Důvod je prostý, lidské vnímání obrazu je „pravidelné“, je vhodné toto vnímání napodobit i v algoritmech řešících zpracování obrazu. Použijeme tedy pravidelnou mřížku, zbývá nám pouze vybrat konkrétní tvar mřížky. Ten se charakterizuje pomocí základního elementu, jehož opakováním vznikne výsledná mřížka. V praxi se používají tři typy základních elementů – trojúhelník, obdélník a šestiúhelník. Trojúhelníková mřížka (Obrázek 5 vpravo) se používá pouze ve speciálních případech, nevýhodou je totiž alternace trojúhelníku postavených na hranu a na špičku. V některých případech může být ale tato struktura vhodná (záleží na snímané oblasti), jedná se ale speciální případy. Nejběžněji používaný element je obdélník (Obrázek 5 vlevo). Hlavním důvodem je jednoduchá výroba snímacích prvků ve tvaru obdélníku, navíc výstup z těchto snímačů vypadá „přirozeně.“ Nevýhodou tohoto elementu je možná deformace hran, které jsou čistě vertikální či horizontální (za předpokladu vodorovné orientaci snímače). Tento problém řeší hexagonální element (Obrázek 5 uprostřed). Pokud se blíže podíváme na mřížku tvořenou šestiúhelníky, lze si všimnout, že možnost, že by hrana

procházela mezi vazebními body, je o hodně menší než u ostatních mřížek. Nevýhodou je složitost základního elementu, v praxi se tedy setkáme nejčastěji s obdélníkovou mřížkou. Proto byla v této práci zvolena obdélníková mřížka.



Obrázek 5 Vzorkovací mřížky - čtvercová, hexagonální, trojúhelníková

Zvolili jsme tedy typ mřížky, nyní se hodí uvést, co mřížka znamená pro vlastní výpočet. Dominantním prvkem mřížek jsou plochy, takže na první pohled se může zdát, že se budeme snažit nějak charakterizovat tvar jednotlivých ploch. Tvary ploch jsou ale pouze důsledkem rozložení bodů na ploše. Nás totiž zajímá rozložení bodů, nikoliv tvar výsledných ploch. Ty jsou pouze důsledkem rozmístění bodů a slouží hlavně jako vizuální pomůcka při rozmísťování bodů na plochu. Každou mřížku lze popsat pomocí matematické funkce, která nám postupně vypočte body ve vymezeném prostoru. Konkrétně si rozebereme funkci pro obdélníkovou mřížku, postup u ostatních mřížek bude pouze nastíněn.

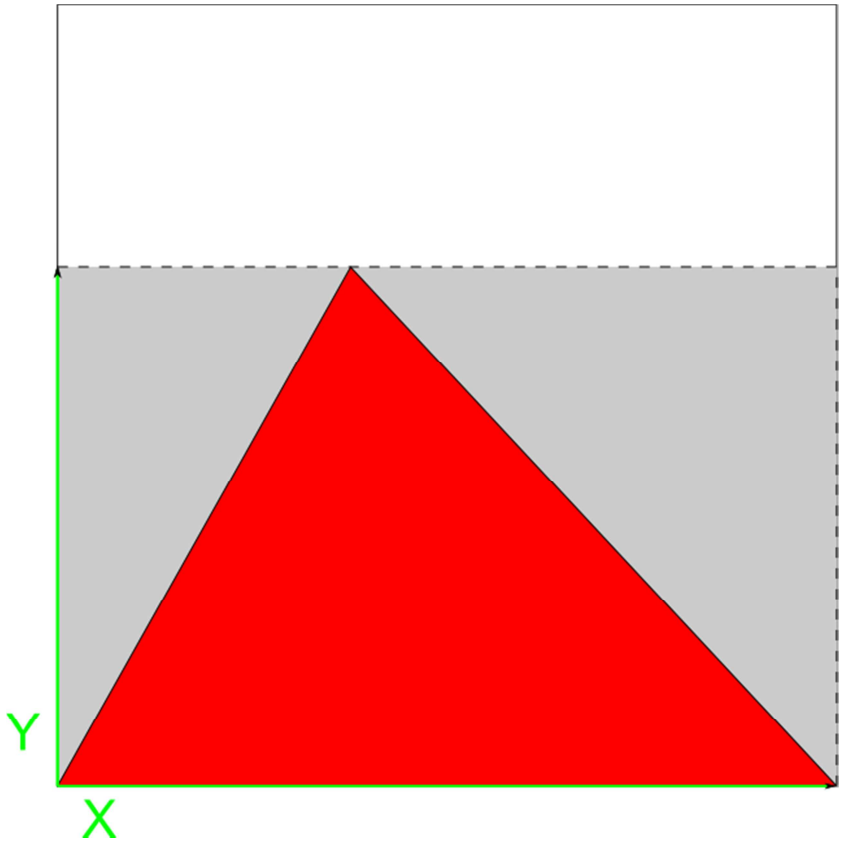
Pro výpočet budeme potřebovat znát souřadný systém oblasti, kterou budeme diskretizovat. Samozřejmě budeme také potřebovat souřadnice bodů omezujících danou oblast. Tyto údaje v našem případě získáme z obalového obdélníku, který jsme vypočítali v minulém kroku. Osu X tvoří body 1 a 2, osu Y poté body 1 a 3. Získané vektory nemusíme normalizovat, protože je nebudeme používat jako osy souřadného systému, jejich rolí bude pouze charakterizovat přírůstky v daných směrech. Pro úspěšnou diskretizaci prostoru je nutné zvolit jeden parametr, a tím je hustota diskretizační sítě. Typicky je charakterizována jedním číslem, které říká, kolika body se rozdělí daná oblast v jednom směru. Označme toto číslo jako **n**. Pokud bychom se nacházeli v jednorozměrném prostoru (úsečce) a použili jsme diskretizaci s hustotou **n**, získali bychom úsečku rozdělenou na (**n** - 1) oblastí, ohraničených **n** body. Ve 2D prostoru (ploše) bychom pak získali (**n** - 1) * (**n** - 1) plošek spolu s n^2 body.

V našem případě se sice nacházíme v 3D prostoru, body trojúhelníku ale leží v jedné rovině, můžeme tedy použít diskretizaci na 2D ploše. Potřebujeme k tomu pouze souřadný systém dané plochy, který jsme získali z obalového obdélníku.

Nyní můžeme přistoupit k vlastnímu rozmístění bodů na plochu. Rozmístění je v mřížce pravidelné jak ve směru osy X, tak ve směru osy Y. První řešení, které se nabízí, je vzít osy obdélníku, vydělit je n, čímž bychom získali přírůstky v jednotlivých směrech a dopočítat tak potřebné body. Bohužel výsledná textura je čtvercová, jsme tedy nuceni vsadit obdélník do čtverce. Pokud nechceme, aby byla výsledná textura v jednom směru deformována, musíme stanovit jednotný diskretizační krok jak pro směr osy X, tak osy Y. Obrázek 6 ukazuje situaci, kde jsme vsadili obdélník do čtverce a strana ve směru osy X je delší než strana ve směru osy Y. Musíme tedy uměle nastavit obdélník ve směru osy Y tak, abychom dostali čtverec. Umožní nám to uložit výsledné body do textury (čtverce). Bohužel budeme nuceni počítat zbytečné body (ty které leží nad trojúhelníkem). Nastavení obdélníku lze realizovat pomocí jednotné délky kroku jak ve směru osy X, tak ve směru osy Y. Otestujeme tedy, která strana obdélníku je delší a tu vezmeme jako výchozí stranu. Vektor této strany pak vydělíme **n**, čímž získáme přírůstkový vektor v daném směru. Přírůstek o stejné délce musíme provádět také ve směru druhé osy. Normalizujeme tedy vektor druhé osy a vynásobíme ho délkou prvního přírůstkového vektoru. Takto získáme dva vektory, které charakterizují přírůstky v obou osách.

Poté, co získáme přírůstkové vektory, můžeme přistoupit k vlastnímu výpočtu jednotlivých bodů. Jak bylo zmíněno výše, obdélníková mřížka má body rozmístěny rovnoměrně, výpočet bodů tedy realizujeme pouhým přičítáním přírůstkových vektorů. Vezmeme první bod obalového obdélníku (bod 1) a postupně k němu budeme přičítat přírůstkový vektor ve směru osy X (celkem **n-1** krát), čímž získáme první řádek bodů diskretizovaného prostoru. Pro výpočet bodů v dalším řádku přičteme k bodu č.1 přírůstek ve směru osy Y a zopakujeme přičítání ve směru osy X. Tento postup opakujeme, dokud kompletně nepokryjeme danou oblast body.

Tento postup je tzv. „naivní“, počítáme zde vše, co nám přijde „pod ruku“. Pro urychlení výpočtu je možné počítat pouze body, které se nacházejí uvnitř rekonstruovaného trojúhelníku, ostatní pak můžeme vynechat, protože se během dalších výpočtů nemusí použít.



Obrázek 6 Umístění trojúhelníku na texturu

Tomuto urychlení se nyní budeme věnovat blíže. Konkrétně se budeme věnovat tomu, jak zjistit, jestli se bod nachází uvnitř či vně trojúhelníku. Vzhledem k potenciálně vysokému počtu bodů (v řádu desetitisíců až statisíců), musíme použít řešení, které je výpočetně co nejefektivnější. Rozebereme si zde tři různé metody, kde každá využívá naprosto odlišný přístup.

První metodou je porovnávání obsahů trojúhelníků, které bylo již zmíněno v kapitole „Nalezení nejvhodnější fotografie.“ V případě fotografie se pracovalo ve 2D prostoru, nyní potřebujeme vzorec pro 3D. To ale na vzorec nemá příliš velký vliv. Jedinou změnou je jiný výpočet délky jednotlivých stran, což je ale naprosto minimální komplikace. Nabízí se tedy znovu využití tohoto algoritmu, protože bychom ušetřili čas při implementaci. Navíc bychom měli jistotu, že algoritmus bude fungovat, protože je již otestován. Detekce pomocí porovnávání obsahů má ale jednu nevýhodu, a tou je rychlost (či spíše pomalost) algoritmu. Pro každý bod jsme nuceni počítat délku tří přímek (spojení s body A, B, C – viz. Obrázek 1). Navíc vlastní vzorec pro výpočet obsahu využívá odmocninu, jejíž výpočet je z hlediska výpočetního času velice náročný. Tato vlastnost zmíněný algoritmus více méně diskvalifikuje, protože ostatní algoritmy nabízejí efektivnější výpočet.

Druhý přístup využívá vektorového počtu. Výpočet se děje pomocí násobení vektorů, a to jak pomocí vektorového, tak pomocí skalárního součinu vektorů. Tyto operace myslím není třeba na tomto místě uvádět, jedná se o triviální operace, které tvoří základní funkce vektorového počtu. V [Přívratská2007] lze případně nalézt zavedení těchto operací. Myšlenka této metody je velice triviální – bod X se musí nacházet na stejné straně vektorů, které tvoří trojúhelník ABC (značení opět viz. Obrázek 1). Problémem může být, jak zjistit na které straně má bod X ležet vůči daným vektorům. Můžeme si ale všimnout, že bod X musí ležet na stejné straně jako zbývající bod trojúhelníku (pokud hledáme stranu u **AB**, použijeme bod C jako referenci). Musíme tedy porovnat, jestli vektory vytvořené vektorovými součiny ukazují stejným směrem. Pro toto porovnání použijeme výše zmíněný skalární součin. Výsledek skalárního součinu lze použít pro výpočet velikosti úhlu, který dané dva vektory svírají. Pro náš případ stačí vědět, že pokud skalární součin je větší nebo roven nule, pak vektory ukazují stejným směrem (stejným směrem znamená, že úhel, který svírají, je maximálně 90°). Nyní si daný postup rozepíšeme matematicky pro stranu AB. Jak je vidět v rovnici **(4)**, vzorec pro výpočet čísla n, které charakterizuje úhel mezi vektory (není to přímo velikost úhlu) se získá pomocí tří součinů, přičemž dva z nich jsou vektorové („×“ značí vektorový součin, „∗“ pak skalární).

$$\boldsymbol{n} = (\boldsymbol{AB} \times \boldsymbol{AX}) * (\boldsymbol{AB} \times \boldsymbol{AC}) \qquad \textbf{(4)}$$

Tento výpočet tedy zopakujeme pro zbývající dvě strany trojúhelníku, a pokud budou všechna n větší než nula, pak se bod nachází uvnitř trojúhelníku. Jak je vidět, výpočet je velice jednoduchý, rychlost výpočtu je také velice slušná. Přesto tento postup není využit, protože se podařilo implementovat ještě efektivnější metodu.

Metoda, která byla nakonec použita v této práci, využívá pro rozhodování barycentrické souřadnice. Ty charakterizují pozici bodu v n-úhelníku pomocí vážené kombinace bodů, které tvoří daný n-úhelník. Tyto souřadnice si lze také představit jako zavedení nového souřadného systému, kdy pozici bodu X charakterizujeme kombinací vektorů **AB** a **AC**, přičemž bod A tvoří počátek souřadného systému. Výpočet polohy bodu X tedy můžeme charakterizovat rovnicí **(5)**.

$$\boldsymbol{X} = A + u * \boldsymbol{AB} + v * \boldsymbol{AC} \qquad \textbf{(5)}$$

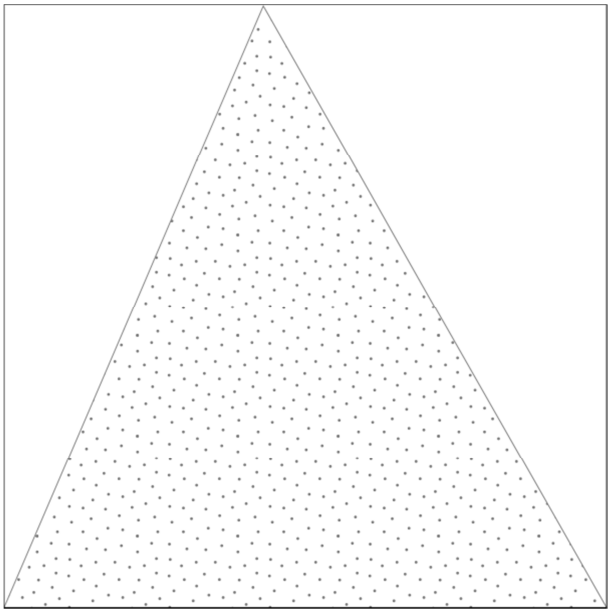
Pro nalezení koeficientů **u** a **v** potřebujeme dvě rovnice, které získáme skalárním vynásobením rovnice **(5)** vektorem **AB**, respektive **AC**. Po úpravě získaných rovnice a vyjádření koeficientů získáme vztahy pro výpočet **u (6)** a **v (7)**.

$$\boldsymbol{u} = \frac{(AB*AB)*(AX*AC)-(AB*AC)*(AX*AB)}{(AC*AC)*(AB*AB)-(AC*AB)*(AB*AC)} \quad \textbf{(6)}$$

$$\boldsymbol{v} = \frac{(AC*AC)*(AX*AB)-(AC*AB)*(AX*AC)}{(AC*AC)*(AB*AB)-(AC*AB)*(AB*AC)} \quad \textbf{(7)}$$

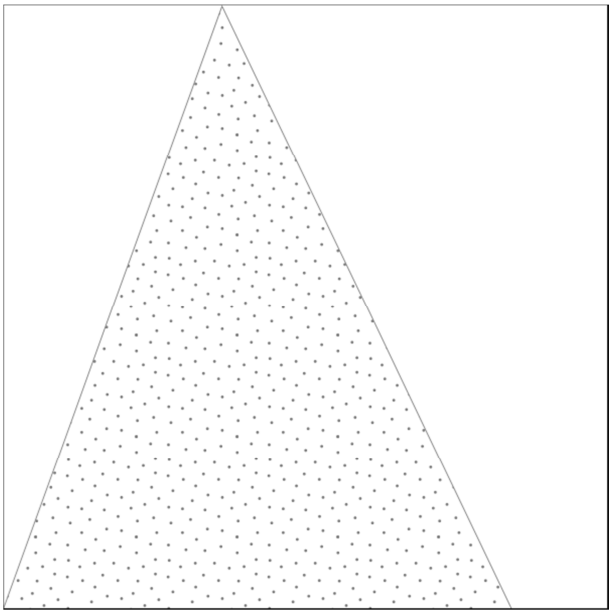
Pokud bod X leží uvnitř trojúhelníku ABC, pak koeficienty musí být kladné a jejich součet musí být menší nebo roven 1. Výpočet se může zdát složitější oproti předchozí metodě, je nutné si ale uvědomit, že vzorce pro **u** a **v** se počítají pro každý bod pouze jednou. Ve výsledku se tedy jedná o nejefektivnější metodu. Tato metoda tedy byla zvolena jako metoda detekce polohy bodu vůči danému trojúhelníku.

Eliminací bodů, které neleží uvnitř rekonstruovaného trojúhelníku, se dosáhne velice dobrého zefektivnění výpočtu, protože se rekonstruuje barva pouze poloviny a méně bodů (polovina za předpokladu, že trojúhelník lze přesně zasadit do čtverce, v ostatních případech se počet bodů sníží ještě více). Obrázek 7 ukazuje případ, kdy je spodní strana trojúhelníku stejně dlouhá jako jeho výška. To znamená, že trojúhelník lze přesně umístit do čtverce (za předpokladu, že horní bod trojúhelníku leží mezi dvěma spodními body). V tomto případě optimalizací snížíme počet bodů přesně na polovinu. Strana čtverce je totiž dlouhá jako spodní strana trojúhelníku a zároveň jako výška trojúhelníku. Obsah trojúhelníku lze vypočítat jako součin délky strany a výšky příslušné strany dělená dvěma. Po dosazení je tedy vidět že trojúhelník zabere přesně polovinu obsahu čtverce.



Obrázek 7 "Čtvercový" trojúhelník

Častěji ale nastane případ, kdy trojúhelník nelze přesně zasadit do čtverce. Tento případ ukazuje Obrázek 8. Z obrázku je patrné, že červená plocha v tomto případě zabírá ještě méně než polovinu plochy čtverce, takže výsledný poměr počítaných bodů vůči celkovému počtu bodů ve čtverci bude ještě menší než jedna polovina.



Obrázek 8 "Obdélníkový" trojúhelník

1.5.Zpětná projekce bodů rastru na fotografii

Posledním krokem výpočtu je nalezení polohy bodů (tvořících obdélník / trojúhelník v prostoru) na fotografii a tím pádem jejich barvu. Jedná se o podobnou úlohu, kterou realizuje fotoaparát při zachytávání fotografie. Při pořizování fotografie se na snímači fotoaparátu zachycuje barva objektu, jež leží na polopřímce, která vychází z ohniska optické soustavy fotoaparátu a prochází skrz jeden z pixelů tvořících matici pixelů snímače. Samozřejmě paprsky z fotoaparátu nevycházejí, naopak vycházejí z předmětů a míří směrem k fotoaparátu. Naším úkolem je tedy tuto úlohu realizovat matematicky. Vstupem bude poloha bodu v prostoru, výstupem pak souřadnice bodu na fotografii (konkrétně nás zajímá barva bodu).

Prvním krokem je zjistit, která data jsou potřeba k úspěšné kalkulaci výsledné barvy bodu. Určitě potřebujeme souřadnice bodu v prostoru. Abychom věděli, kam barvu bodu uložit, potřebujeme jeho polohu ve čtverci textury (neplést s polohou na fotografii). Posledním (a nejdůležitějším) údajem, který potřebujeme, jsou informace o kameře. Ty se dělí na dva typy – vnější a vnitřní. Mezi vnější údaje se řadí informace o poloze a natočení – říká se jim vnější, protože je můžeme vidět. Vnitřní parametry jsou pak charakteristiky optické soustavy fotoaparátu. Hlavním parametrem v této skupině je ohnisková vzdálenost. Lze zde zahrnout i vady optické soustavy jako nečtvercovitost pixelů, zkreslení čoček a podobně. V této práci se zohledňuje pouze ohnisková vzdálenost. Další parametry byly vynechány, protože je velmi obtížné zjistit je automaticky a není dost dobře možné po uživateli požadovat, aby tyto parametry zjišťoval a zadával do programu. Mezi parametry se také řadí rozlišení fotografie, to se ale obchází normováním rozměru fotografie do rozsahu <0, 1>. Výpočet probíhá v tomto rozsahu a až výsledná pozice se převede pomocí rozlišení konkrétní fotografie na souřadnice na fotografii. Pokud vše shrneme, potřebujeme souřadnici bodu v prostoru, souřadnici na textuře a data o kameře.

Výpočet si rozdělíme na dvě části – nalezení průsečíku polopřímky s plochou snímače fotoaparátu a určení polohy pomocí X a Y vektorů fotoaparátu. Pro první část použijeme pozici bodu v prostoru, pozici kamery a vektor, který charakterizuje směr, kam kamera míří (tento se běžně nazývá „view“ vektor). Úkolem této části je vypočítat bod, který vznikne jako průsečík polopřímky jdoucí z bodu trojúhelníku do kamery a plochy snímače fotoaparátu. Vektor získáme jednoduše odečtením bodů. Plochu v prostoru lze charakterizovat několika způsoby, v našem případě byl zvolen popis

pomocí normálového vektoru a jednoho bodu ležícího na ploše. Tento popis byl zvolen vzhledem k dostupným datům – bod ležící na ploše je poloha fotoaparátu v prostoru, normálový vektor je pak view vektor fotoaparátu. Pro výpočet bude použito následující značení – A je počáteční bod vektoru (jdoucího z bodu trojúhelníku), B je bod ležící na vektoru (určuje směr vektoru). **N** je vektor normály plochy, P pak bod ležící na ploše (v našem případě se používá poloha fotoaparátu). Poloha průsečíku se počítá jako podíl vzájemných úhlů vektorů **AP** s **N** a **AB s N**. Výsledkem je číslo, jímž „prodloužíme" vektor **AB**, kde poté bod B bude hledaný průsečík. Ve vzorci je poloha průsečíku značena jako **I**, násobení dvou vektorů charakterizuje skalární součin vektorů, součin skaláru a vektoru znamená vynásobení všech složek vektoru daným číslem.

$$u = \boldsymbol{N} * \boldsymbol{AP} / \boldsymbol{N} * \boldsymbol{AB}$$

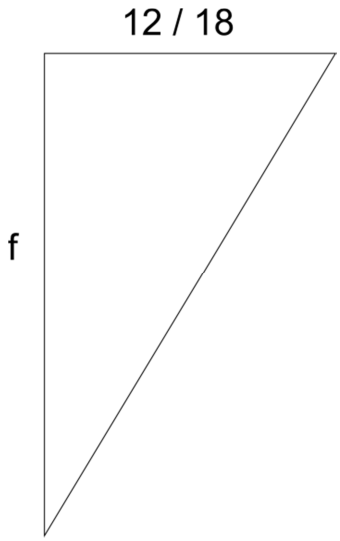
$$I = A + u * \boldsymbol{AB}$$

Výsledkem tohoto výpočtu je tedy poloha bodu na snímači fotoaparátu. Bod **I** charakterizuje absolutní hodnotu průsečíku v prostoru, ta nás ale nezajímá. My potřebujeme znát relativní polohu vůči středu projekce, abychom mohli zjistit pozici na fotografii. Vypočteme tedy vektor **PI**, který směřuje ze středu fotografie do průsečíku. Poloha na fotografii má dvě souřadnice a my tedy potřebujeme také charakterizovat polohu průsečíku pomocí dvou čísel. K tomu využijeme zbývající údaje o kameře – dva vektory charakterizující natočení snímače. Ty si můžeme představit jako bázové vektory souřadné soustavy snímače (vektory os X a Y, podobně jako v případě obalování trojúhelníku do obdélníku). Úkolem je vypočítat projekce vektorů na dané osy. Pokud si osu označíme **X** (předpokládáme normalizovaný vektor – jeho délka je rovna jedné), vstupní vektor **V** a výslednou projekci $\mathbf{V_X}$ pak vzorec pro projekci vypadá následovně –

$$\boldsymbol{V_X} = \boldsymbol{X} * (\boldsymbol{X} * \boldsymbol{V})$$

Takto rozložíme vektor na dvě části, jednu ve směru osy X, druhou ve směru osy Y (nejedná se o osy globálního souřadného systému, ale o osy snímače fotoaparátu). Posledním krokem je převedení těchto vektorů na souřadnice na fotografii. K tomu využijeme vnitřní parametr kamery, ohniskovou vzdálenost. Ohnisková vzdálenost určuje, jak fotoaparát vidí do šířky (a samozřejmě i do výšky), určuje totiž zorný úhel fotoaparátu. Čím je ohnisková vzdálenost menší, tím je snímač blíže ohnisku a paprsky jdoucí z ohniska snímačem pokryjí větší plochu v prostoru. Díky ohniskové vzdálenosti jsme schopni spočítat krajní vektory ve směr os X a Y.

Ohnisková vzdálenost získaná z fotografie je vztažena vůči běžnému kinofilmovému políčku (rozměry 36 x 24 mm). Pro zjednodušení výpočtu by ale pro nás bylo vhodnější, abychom převedli tento problém na jiný problém, kde je ohnisková vzdálenost rovna jedné a tím pádem by rozměry snímače (nyní již virtuální) šlo jednoduše převádět na souřadnice na fotografii. Obrázek 9 představuje náčrtek situace. Spodní bod tvoří ohnisko snímače, vrchní čára je pak polovina snímače. Čísla představují rozměr snímače (výška nebo šířka), navíc jsou dělena dvěma, aby odpovídala obrázku. S dostupných čísel můžeme dopočítat úhel u spodního bodu dle vzorce $\alpha = \tan^{-1}\frac{a}{f}$, kde ***a*** je 12 nebo 18. Přepočet na trojúhelník s ***f*** rovno 1 je poté pouze otázkou vyjádření ***a***.



Obrázek 9 Trojúhelník ohnisková vzdálenost / snímač

Poměr délky těchto krajních vektorů (**a** pro výšku nebo šířku) s délkou vektorů rozkladu průsečíku pak již tvoří relativní souřadnice na fotografii. Samozřejmě se musí zohledňovat rozlišení fotografie a jiná orientace souřadné soustavy, to ale bude blíže rozebráno v části věnující se implementaci tohoto výpočtu.

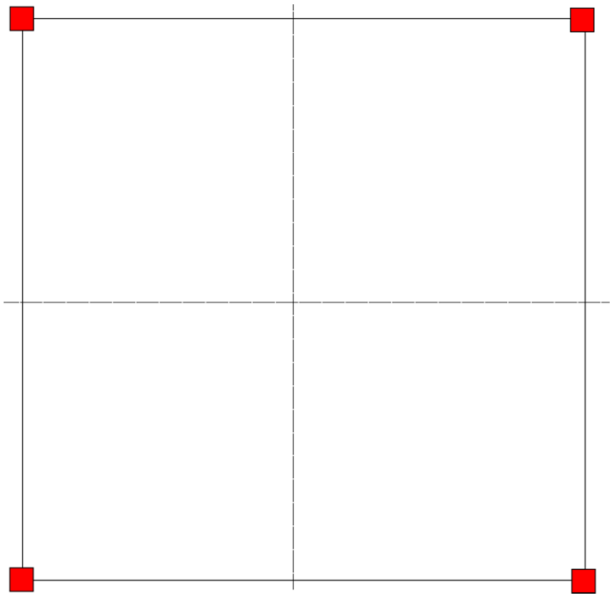
Výsledná poloha bodu na fotografii je obecně charakterizována dvěma neceločíselnými čísly. Pro získání výsledné barvy je tedy nutné rozhodnout, z kterého bodu na fotografii či z jaké kombinace bodů se vytvoří výsledná barva bodu na textuře. Tuto úlohu řeší metody interpolace. Celkem byly implementovány dvě metody interpolace – interpolace nejbližším sousedem a bilineární interpolace. Kromě těchto dvou metod se běžně také využívá kubická interpolace, v této práci ale nebyla

implementována (v implementaci bude zmíněno, že je možno přidávat další metody bez větších komplikací).

Zjednodušená formulace úlohy interpolace je aproximace obecně spojité funkce pomocí funkce nespojité. Postupy metod interpolace se běžně definují pro jednorozměrné úlohy, rozšíření na n-rozměrné úlohy je pak triviální opakování postupu řešení jednorozměrné úlohy i na další rozměry.

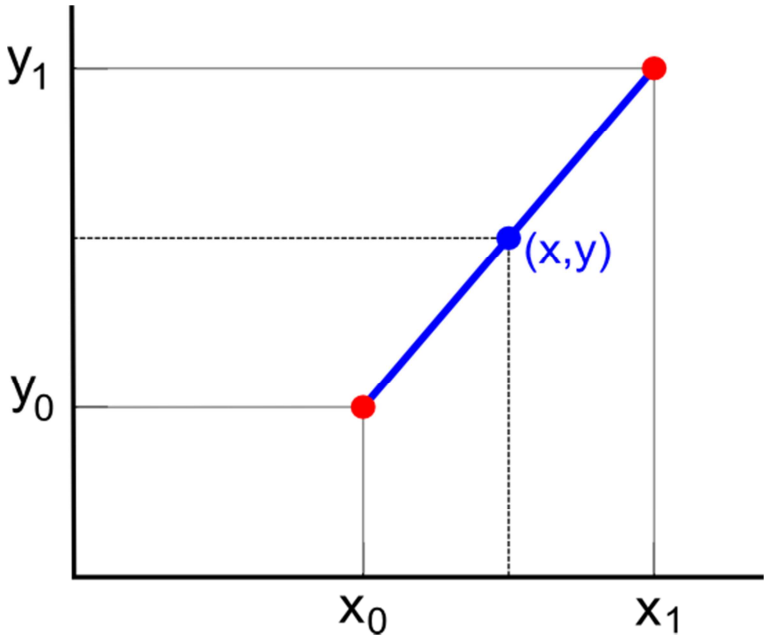
První použitou metodou interpolace je metoda interpolace nejbližším sousedem. Jak již název metody napovídá, hledá se bod diskrétní funkce, který je nejblíže bodu funkce spojité (funkce se na sebe přikládají bez jakékoliv úpravy – posunu, inverze apod., bod se souřadnicí 1.0 ve spojité funkci se kryje s bodem 1 funkce diskrétní). Pro hledání nejbližšího bodu se typicky používá funkce zaokrouhlení reálného čísla. Výsledek zaokrouhlení je pak již souřadnice bodu diskrétní funkce, jehož barvu použijeme jako barvu výsledného bodu na textuře. Je patrné, že tato metoda je velice jednoduchá, její implementace je pak velice efektivní a rychlá, protože je nutné vykonat pouze **n** zaokrouhlení pro výpočet polohy bodu, kde **n** značí rozměr interpolované funkce. Daní za rychlost je pak menší kvalita výsledného obrazu. Při zvětšení obrazu (výstupní obraz má vyšší rozlišení než vstupní) dochází ke zvýraznění skoků, je patrné „rozkostičkování“ obrazu. Naopak při zmenšení obrazu může dojít (a ve většině případů dochází) k citelné ztrátě vizuální informace. I při nepříliš velkém zmenšení obrazu dochází k poškození či ztrátě tenkých čar. I když se často jedná o ztrátu pár pixelů (v řádu jednotek), výsledný vzhled obrazu je degradován často velmi citelně (lidský mozek se soustředí hlavně na hrany či hranice objektů).

Obrázek 10 ukazuje situaci, kdy máme na ploše 4 body (diskrétní 2D funkce) a my potřebujeme rozhodnout, ke kterému bodu přiřadit vstupní souřadnice (ty se nacházejí uvnitř vyznačeného čtverce). Na obrázku se nacházejí dvě přerušované čáry, které dělí čtverec na 4 díly. Tyto čáry představují limitní pozice, ve kterých se mění výsledný přiřazený bod. Pokud se tedy vstupní body nacházejí v levé horní části, pak výsledkem je horní levý bod. V případě, že bod leží na jedné či více čarách (přerušovaných i plných), pak záleží na konkrétní implementaci zaokrouhlování, typicky se ale použije nejbližší „vyšší“ uzel (uzel, jehož souřadnice je numericky větší).



Obrázek 10 Interpolace nejbližším sousedem

Druhou metodou interpolace je interpolace lineární (v našem případě bilineární, nacházíme se ve 2D prostoru). Jak je z názvu metody patrné, bude se využívat lineární kombinace okolních bodů. Postup si ukážeme pro 1D případ, pro 2D příklad bude uveden pouze finální vzorec. Obrázek 11 je nákres 1D situace. Pro výpočet potřebujeme souřadnice počítaného bodu (necelé číslo, v obrázku jako **x**), souřadnice okolních bodů (celá čísla, v obrázku $\mathbf{x_0}$ a $\mathbf{x_1}$) a příslušné barvy okolních bodů ($\mathbf{y_0}$ a $\mathbf{y_1}$).



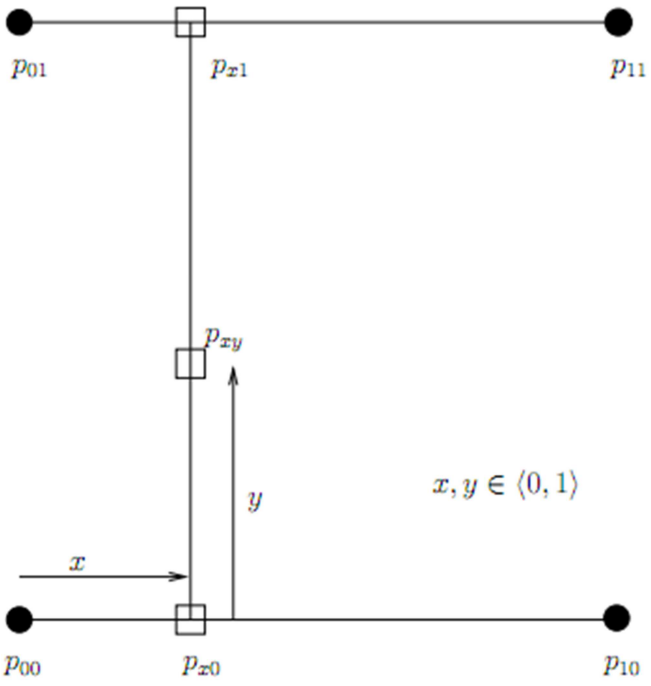
Obrázek 11 Lineární interpolace

K výpočtu **y** (výsledná barva bodu) by se nám hodila rovnice charakterizující modrou křivku. Závislost x a y lze geometricky odvodit z obrázku jako

Nás zajímá y, takže ho z rovnice vyjádříme a získáme rovnici

Tím jsme získali vztah, podle kterého lze zjistit hodnotu y pro bod, který leží mezi dvěma známými body. Jak bylo uvedeno výše, interpolační metody se typicky zavádí pro 1D úlohy a pro vícerozměrné metody se daný postup aplikuje postupně pro všechny rozměry úlohy. Pro bilineární interpolaci si to lze představit tak, že nejdříve se posuneme ve směru x z levého spodního bodu, následně se posuneme ve směru y, čímž dojdeme na žádané místo (pro lepší představu se lze podívat na Obrázek 12). Délky těchto dvou cest se pak využijí pro výpočet. Ve výsledku pak získáme vztah

(f značí hodnotu v daném bodě, x a y jsou pak délky jednotlivých cest). Pomocí této rovnice jsme tedy schopni dopočítat barvu všech bodů tvořících texturu (ať už to jsou všechny body textury nebo pouze ty „užitečné“).

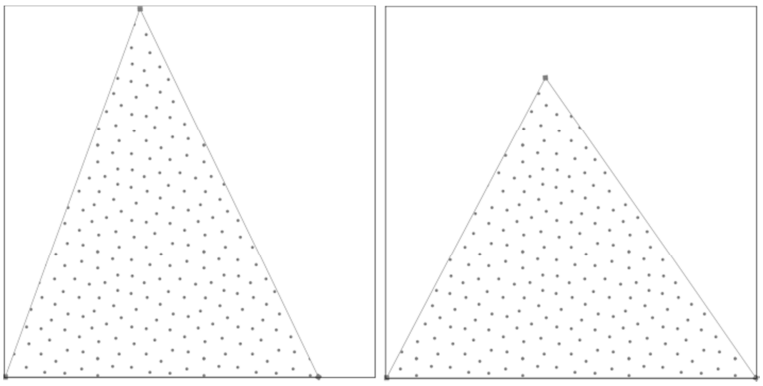


Obrázek 12 Bilineární interpolace

1.6.Výpočet souřadnic textury

V tomto momentě máme k dispozici již rekonstruovanou texturu, která je uložena v příslušném datovém úložišti. Nastává však ještě jedna komplikace a to, že ukládáme trojúhelník do čtverce, takže je nutné spočítat a uložit souřadnice vrcholů trojúhelníku na textuře.

Základní orientace trojúhelníku je vždy stejná, první dva body tvoří spodní hranu textury, třetí bod se pak nachází nad nimi. Pro úspěšný výpočet polohy bodů na textuře potřebujeme zjistit dvě charakteristiky trojúhelníku – jestli je vyšší než širší (či naopak) a také v jaké poloze vůči základně se nachází třetí bod. To, jestli je vyšší nebo širší, lze určit ze souřadného systému obalujícího obdélníku, protože uchováváme nenormalizované osy (což jsou vlastně strany obdélníku). Stačí tedy porovnat délky těchto stran. Obrázek 13 ukazuje obě možné situace. Tento údaj říká, zda druhý a třetí bod budou ležet na samém okraji textury. První bod leží v počátku souřadného systému textury (jak bude ukázáno dále, nemusí to vždy platit), druhý bod leží na pravém okraji v případě, že trojúhelník je širší než delší a třetí bod leží na vrchním okraji v opačném případě. Vzdálenost bodu od okraje (pokud na něm tedy neleží) lze pak zjistit z poměru délek stran obalujícího obdélníku. Samozřejmě může nastat případ, kdy všechny body leží na hranách textury, pak není nutné nijak měnit souřadnice textury z důvodu větší šířky / výšky.



Obrázek 13 Vyšší (vlevo) a širší trojúhelník

Druhou charakteristikou je poloha třetího bodu vůči základně. Jak ukazuje Obrázek 3 (na straně 15), jsou možné celkem tři polohy. Zjištění polohy je vysvětleno v kapitole „1.3 Příprava omezujícího obdélníku“ (začátek kapitoly strana 14), zde si tedy uvedeme pouze důsledky polohy bodu na výpočet. Podobně jako předchozí charakteristika také poloha třetího bodu mění, jestli bod bude nebo nebude ležet na okraji textury. U šířky / výšky trojúhelníku se měnila horizontální poloha druhého bodu

a vertikální poloha třetího bodu. U relativní polohy bodu se mění pouze horizontální polohy bodů. V případě, že třetí bod leží nalevo od prvního, první bod neleží v počátku souřadného systému textury, ale je posunut doprava a třetí bod leží na levém okraji textury. Pokud třetí bod leží mezi prvním a druhým bodem (myšlena projekce bodu na osu tvořenou prvním a druhým bodem), nedochází k žádné změně. A pokud se třetí bod nachází napravo od druhého, druhý bod se posouvá doleva a třetí bod leží na pravém okraji textury.

Díky těmto dvěma charakteristikám určíme velkou část souřadnic – body ležící na okraji a jejich polohu z poměru délek stran. Zbývající souřadnice (horizontální polohy některých z bodů) se pak dopočítá z vhodného poměru délek projekce vektoru **AC** na osu **X** a délky vektoru **AB** (bod A odpovídá prvnímu bodu, B druhému a C třetímu). Ke každé souřadnici musíme samozřejmě uchovávat informaci, ke kterému bodu trojúhelníku patří. Získáme tedy tři dvojice údajů souřadnice bodu / id bodu, které budeme používat při zobrazování textury či při exportu do externího formátu.

2. Implementace v jazyku Java

V této části se budeme podrobně věnovat algoritmizaci jednotlivých kroků rekonstrukce. Nebude se ale jednat o detailní popis každého řádku kódu, spíše se budeme zabývat zajímavými částmi, případně částmi, jejichž funkčnost by nemusela být na první pohled zřejmá.

Pro snazší orientaci v textu a přehlednost budou názvy tříd zvýrazněny kurzívou (např. „*Vector3D*“), názvy rozhraní budou navíc obaleny menšítkem a většítkem (např. „<*Iterator*>“). Názvy metod pak budou uvedeny tučným písmem (např. „**clear()**“). Pokud metoda vyžaduje parametry, budou buď vypsány v závorce (např. „**delete(A)**“ – smazání prvku A), případně budou v závorce uvedeny tři tečky (v případě, že parametry nejsou pro potřeby textu důležité, např. „**intersectionSkew(…)**“).

Jednotlivé třídy a jejich metody lze nalézt v přiložených zdrojových kódech. Pokud nebude uvedeno jinak, třídy vytvořené během této diplomové práce se budou nacházet v balíčku „cz.tul.engine.textures“. U jednotlivých tříd nebude zmiňováno přesné umístění, vyhledávat dle názvu dnes umí prakticky každý editor, navíc struktura tříd není příliš složitá. Pokud by případný zájemce neměl k dispozici nějaký takovýto editor, je možné použít klasické vyhledávání v operačním systému, kde si může najít konkrétní umístění třídy, nalézt metodu pak dle názvu umí i obyčejný textový editor.

2.1.Stav aplikace „E3Dm“

Není zde třeba kompletně popisovat strukturu a funkčnost programu E3Dm, popis funkce lze nalézt v [Ječmen2009], konverzi programu do jazyka Java pak v [Ječmen2010], kde jsou popsána také vylepšení výpočetního algoritmu oproti původní verzi. V této části se budeme soustředit hlavně na prostředky a informace, které jsou uloženy v programu, a můžeme je využít pro rekonstrukci.

První sada informací je od uživatele. Jedná se o pozici bodů na fotografii, jejich provázání mezi fotografiemi (přiřazení jednoznačného ID bodu napříč fotografiemi) a soubor trojic bodů tvořících trojúhelníky, ze kterých je model složen. Data o bodech si uchovává každá fotografie sama, úložiště fotografií je pak přístupné pomocí třídy *DataContainers*. Tato třída je implementována jako návrhový vzor Singleton [Kraval2002], je tedy dostupná z libovolné části programu, navíc její implementace je synchronizovaná, takže k ní můžeme přistupovat z více vláken najednou. Informace o polygonech jsou odděleny od fotografií, protože platí obecně pro všechny fotografie. U každého polygonu je také uvedeno, jestli se jedná o „viditelný“ polygon, což ukazuje, jestli polygon je zahrnut ve výsledném modelu a tím pádem jestli bude vidět. Pokud vidět nebude, nemusíme se zabývat rekonstrukcí jeho textury. Data polygonů jsou opět dostupná pomocí třídy *DataContainers*.

Druhou část informací pak tvoří data získaná během vlastní rekonstrukce drátěného modelu. Hlavní část tvoří data o kamerách – jejich vzájemná pozice a orientace. Informace o kameře je tvořena celkem čtyřmi vektory – jeden je polohový, tři popisují natočení kamery. Tímto lze plně popsat polohu a orientaci pohledu v prostoru. Data kamer se neukládají, není tedy nutné, aby byli globálně přístupné. Výpočetní engine si je ale samozřejmě během výpočtu uchovává, můžeme je tedy pro rekonstrukci využít. Rekonstrukce textur následuje ihned po rekonstrukci drátěného modelu, stačí tedy předat data kamer objektu, který bude realizovat rekonstrukci textur.

Poslední využitelnou informací jsou finální pozice bodů modelu v prostoru. Ty mohou ulehčit výpočet, jak ale bude v další části práce uvedeno, není zcela nezbytné je využít. Pro uchování dat o poloze bodů v prostoru je k dispozici třída *FinalData*, která je podobně jako *DataContainers* přístupná odkudkoliv z programu.

2.2.Nalezení nejvhodnější fotografie

Nalezení nejvhodnější fotografie má za úkol ohodnotit fotografie dle předem daného bodovacího systému a pak vybrat tu, která má ohodnocení nejlepší. Jako základ bodování byla vybrána plocha trojúhelníku na fotografii, protože ta má největší vliv na kvalitu výsledné textury. Pro výpočet obsahu byl implementován Heronův vzorec. Před vlastním výpočtem nesmíme zapomenout zjistit, zda je trojúhelník na dané fotografii vůbec vidět. Pro tento úkol stačí použít již implementované metody třídy fotografie, která vrací instanci bodu dle jeho ID, případně hodnotu **null** pokud se bod na fotografii nenachází. Implementace Heronova vzorce se nachází ve třídě SupportMath. Nevýhodou tohoto vzorce jsou čtyři odmocniny, které mohou výpočet zpomalovat, výhodou je naopak univerzálnost, protože nemusíme počítat výšky ani nic podobného, pouze stačí znát body tvořící trojúhelník.

Získaná plocha tvoří pouze základ hodnocení. To se ve výsledku může o dost snížit, záleží, jestli je trojúhelník zakryt, či ne. První se detekuje „jisté“ zakrytí. To nastává v případě, že uvnitř našeho trojúhelníku se nachází jiný bod. Pro zjištění, jestli se bod nachází uvnitř nebo vně byla implementována metoda využívající obsahu podtrojúhelníků v porovnání s celkovým obsahem. Obsah se opět počítá pomocí Heronova vzorce (nejedná se o real-time výpočet, rychlost tedy není nutně na prvním místě), výsledné porovnání se ale nesmí realizovat jako pouhé porovnání pomocí rovná se. Díky nepřesnostem při výpočtech v plovoucí desetinné čárce může nastat situace, kdy bod leží uvnitř trojúhelníku, rozdíl obsahu ale přesto vyjde nenulový. Pokud si ale tuto hodnotu vypíšeme, zjistíme, že se jedná o naprostou zanedbatelnou hodnotu (řádově 1e-35 a méně, což je téměř minimální hodnota pro položky typu double). Přesto to ale není nula, je tedy nutné porovnat, jestli tato hodnota je menší než nějaký vhodně zvolený práh. V našem případě byl práh zvolen 1e-12, což je dostatečně vysoká hodnota pro detekci nuly, přesto spolehlivě zajistí detekci, jestli už bod neleží mimo trojúhelník. V případě detekce, že bod leží uvnitř trojúhelníku (rozdíl obsahů je nulový), snížíme hodnocení fotografie na nulu, čímž prakticky zamezíme jejímu použití jako zdroje dat pro rekonstrukci textury.

Zakrytí může nastat i v případě, že se žádný další bod uvnitř trojúhelníku nenachází. Je nutné použít další způsob detekce zakrytí, a to pomocí průniku úseček tvořících jednotlivé trojúhelníky. Implementace nalezení průsečíku dvou úseček v ploše lze nalézt ve třídě *SupportMath*, jméno metody je **isLineIntersectingOtherLine(…)**. Této

metodě se předají okrajové body úseček, výstupem je pravdivostní hodnota říkající, jestli se dané úsečky protínají. Prvním krokem detekce je výpočet obecných rovnic úseček (rovnice popisu přímky, pokud se bod nachází na úsečce, se musí zjišťovat dodatečně). Ta má tvar $\boldsymbol{a} * x + \boldsymbol{b} * y + \boldsymbol{c} = 0$. Parametry **a**, **b** určíme z normálového vektoru úseček (odečteme polohy krajních bodů od sebe, prohodíme souřadnice a jednu ze souřadnic vynásobíme mínus jednou), parametr **c** pak dosazením libovolného bodu do rovnice a dopočítáním (k dispozici máme dva body ležící na přímce).

Nejprve otestujeme, jestli nejsou úsečky rovnoběžné. Pro to potřebujeme znát směrnice obou přímek, které získáme jako podíl parametrů **a**, **b**. Pokud se směrnice rovnají, pak jsou přímky rovnoběžné a průsečík neexistuje. Nyní víme, že průsečík existuje, musíme ho tedy najít a zjistit, jestli leží na obou úsečkách (to, že bod leží na přímce, neznamená, že leží také na úsečce, protože ta tvoří jen část přímky). Jedná se o řešení soustavy dvou rovnic o dvou neznámých. Výsledné souřadnice nalezneme tedy vhodným pronásobením / vydělením / odečtením jednotlivých koeficientů. Konkrétní vzorce je vidět ve zdrojovém kódu, případně je lze nalézt v literatuře. Posledním krokem je zjištění, jestli bod leží také na úsečkách. To lze realizovat pomocí čtyř testů, jestli souřadnice leží v rozsahu tvořeném krajními body úsečky (například leží-li x-ová souřadnice průsečíku mezi x-ovými souřadnicemi krajních bodů první úsečky). Pokud všechny testy vyjdou pozitivně (bod leží uvnitř rozsahů), víme, že se úsečky protínají.

Jak bylo zmíněno výše, nelze stoprocentně rozhodnout, jestli se jedná o reálné zakrytí. Díky tomu detekce tohoto zakrytí neznamená vyloučení fotografie z výpočtu. Počet možných zakrytí se sečte, zvýší o jedna a výsledkem se vydělí plocha trojúhelníku (zvýšení o jedna je kvůli nulovému počtu zakrytí).

2.3.Výpočet omezujícího obdélníku

Před vlastním výpočtem obdélníku musíme nalézt polohu bodů trojúhelníku v prostoru. Použití dat finálního modelu není nutno nějak podrobněji rozebírat, jedná se pouze o získání dat a jejich uložení do pole. Postup využívající informace z fotografie je již zajímavější. V principu se jedná o výpočet průsečíku několika polopřímek. Pro naše potřeby byl ale algoritmus modifikován, aby nedocházelo k tak velkému zkreslení geometrie (i za cenu menší nepřesnosti polohy).

Postup výpočtu obecně mimoběžných polopřímek je následující – na polopřímce nalezneme bod, který leží nejblíže k druhé polopřímce, stejný postup opakujeme na druhé polopřímce. Průsečík pak leží v polovině spojnice nalezených bodů. Modifikace algoritmu je prostá, místo nalezení středu budeme počítat pouze krajní body a z nich pak budeme používat pouze ten, který leží na žádané polopřímce. Implementaci této změny lze nalézt ve třídě *VectorMath*, název metody **nearestPointsSkewLines(…)**. Ve stejné třídě se také nachází metoda **intersectionSkew(),** která má za úkol počítat průsečík, zájemce si může metody velice jednoduše porovnat.

Na výpočtu omezujícího obdélníku není implementačně nic moc zajímavého, využívají se metody třídy *Vector3D*, případně *VectorMath*, dle vzorce z teoretické části. Jedinou věcí, která stojí za zmínku, je pořadí uložení bodů ve výsledném poli. Pořadí je definováno zleva doprava a odspoda nahoru, takže první je bod vlevo dole, následuje bod vpravo dole, třetí je pak bod vlevo nahoře a poslední se nachází vpravo nahoře. V principu je pořadí úplně jedno, je ale vhodné ho definovat, abychom nemuseli v dalších částech programu hádat, který bod je první, který druhý a tak dále.

2.4.Rasterizace obdélníku

V teoretické části byl popsán tvar mřížky a výpočet souřadnic jednotlivých bodů. Pro připomenutí to byla obdélníková mřížka a souřadnice bodů so počítají pomocí bázových vektorů obalujícího obdélníku. Uživateli je umožněno si vybrat, jak velikou texturu chce, bylo by tedy vhodné nějak „schovat“ úložiště bodů, abychom nemuseli zjišťovat jeho velikost při procházení. K tomu je v Javě k dispozici rozhraní <*Iterator*> s pouhými dvěma metodami – **hasNext()** a **next()**. Pomocí metody **hasNext()** se dotazujeme, jestli jsou k dispozici další body a metoda **next()** vrací instanci dalšího bodu.

Schování úložiště bylo využito ještě k jedné optimalizaci. Textura může být složena z velikého počtu bodů (v řádech stotisíců až milionů), je tedy velice paměťově náročné uchovávat instance těchto bodů v paměti najednou. Proto byl použit návrhový vzor Flyweight [Kraval2002]. Tento vzor se přímo soustředí na minimalizaci paměťových nároků pro aplikace využívající větší množství objektů se stejnou strukturou. Podmínkou je, že se nesmí používat všechny objekty najednou, což naše aplikace hravě splňuje, protože se projekce počítají postupně. Myšlenka vzoru je, že se použije pouze jedna instance dané třídy, která se bude sdílet. Je ale nutné, aby bylo možné měnit parametry, kterými se jednotlivé objekty mezi sebou liší. Třída *RasterPoint* takovou třídu demonstruje. Jedná se o jednoduchou datovou strukturu, která drží dvě souřadnice a vektor. Souřadnice lze získávat a měnit pomocí dostupných metod, u vektoru se změna dělá pomocí metod třídy *Vector3D*. Máme tedy k dispozici třídu, kterou lze sdílet, a víme, že při používání nebude docházet ke zbytečnému alokování další paměti. Při výpočtu vždy pouze nastavíme nové souřadnice X a Y a posuneme vektor, čímž získáme nový bod na rastru bez jakéhokoliv vytváření nových objektů.

Nyní zmíníme něco málo o vlastním výpočtu polohy bodu na rastru. Tuto činnost realizuje třída *PointRasterIterator*, která implementuje výše zmíněné rozhraní <*Iterator*>. Pro výpočet polohy potřebuje znát souřadnice bodů obalujícího obdélníku a hustotu diskretizační mřížky. Body se předávají v konstruktoru, hustota je pak uložena v nastavení, ze kterých si ji třída načte. Inkrementační vektory v jednotlivých směrech se získají odečtením bodů obdélníku od sebe (prvního a druhého pro osu X, prvního a třetího pro osu Y), získané vektory pak vydělíme hustotou diskretizační mřížky. Výchozí bod je první bod obdélníku, další získáme přičtením jednoho z inkrementačních vektorů k současné poloze. Přičítání opakujeme tolikrát, kolik

je hustota mřížky. Poté se vrátíme zpět na kraj obdélníku (pouze v jednom směru, např. k levému kraji v dané výšce), k získané poloze přičteme druhý inkrementační vektor a pokračujeme ve výpočtu. Toto opakujeme, dokud neprojdeme celou plochu mřížky.

Počet generovaných a tím pádem i počítaných bodů lze redukovat pomocí rozhodování, jestli bod leží uvnitř rekonstruovaného trojúhelníku. Jak bylo zmíněno v teoretické části, byl zvolen postup pomocí výpočtu baricentrických souřadnic. Cílem je vypočítat souřadnice **u** a **v** a pomocí nich rozhodnout, jestli bod leží uvnitř trojúhelníku. Implementace metody rozhodující, jestli bod uvnitř trojúhelníku nalezneme ve třídě *SupportMath*, konkrétně se jedná o metodu **isPointInsideTriangle(A,B,C,X)**. Je zde implementace i pro 2D případ, nás ale zajímá případ ve 3D. Na začátku metody si vytvoříme vektory tvořící náš trojúhelník (stačí vektory **AB** a **AC**) a vektor **AX**. Následuje vypočtení potřebných skalárních součinů, které poté dosadíme do vzorce pro **u** a **v**. Rozhodnutí, jestli bod leží uvnitř trojúhelníku, je pak realizováno testem, jestli **u** a **v** jsou kladné a jestli jejich součet je menší než jedna. Metoda používá pouze metody třídy Vector3D, není tedy důvod detailněji rozebírat jednotlivá volání. Hodí se také zmínit, že výpočet je vcelku rychlý (příprava 3 vektorů, 5x skalární součin vektorů a několik násobení a dělení), úspora času oproti naivnímu přístupu, kdy se počítají všechny body, je veliká.

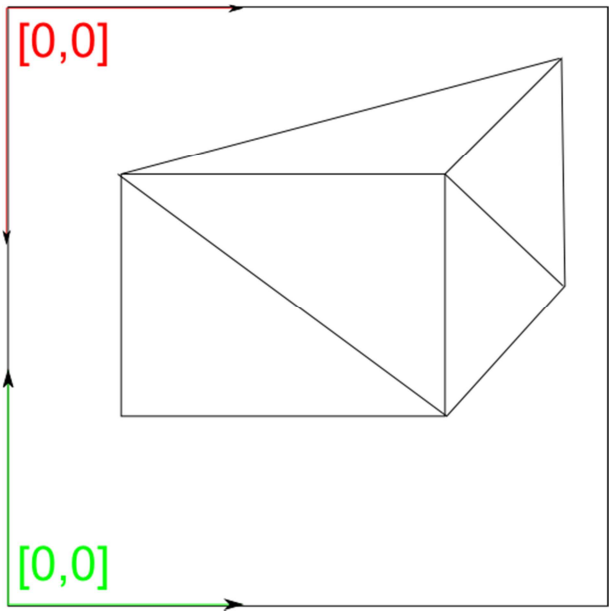
2.5.Zpětná projekce bodů z rastru na fotografii

Projekce bodů na fotografii je posledním krokem vlastní rekonstrukce, další kroky se týkají již pouze reprezentace textury v paměti a uložení dat na disk. Výpočet průsečíku s rovinou snímače je přímočarý, pro výpočet se použije metoda ze třídy *VectorMath*, které předáme polopřímku (její počátek a směrový vektor) a popis roviny (bod ležící v rovině a normálový vektor), metoda nám vrátí instanci třídy *Vector3D*, která popisuje bod průsečíku. Pomocí vypočteného bodu vypočteme vektor charakterizující polohu bodu na fotografii (vektor vychází ze středu snímače). Tento vektor však ještě musíme rozložit na bázové vektory roviny, které získáme z informací o natočení kamery. Projekce vektoru na osu je pak pouhý skalární součin vektoru polohy s osou, výsledek pak určuje délku vektoru ve směru dané osy. Převod na souřadnice na fotografii se pak děje za použití ohniskové vzdálenosti, při které byla fotografie pořízena. Ta se získá z EXIF dat fotografie [Ječmen2009].

Výsledná poloha na fotografii je ale obecně reálné číslo, takže ještě nemůžeme určit výslednou barvu bodu. Pro výpočet musíme použít některou z metod pro interpolaci barvy z okolních bodů. Implementovány jsou dvě metody – metoda nejbližšího souseda a bilineární interpolace. Implementaci lze nalézt v balíčku „cz.tul.engine.textures.interpolation.“ Obě třídy implementují rozhraní <*IInterpolation*>, konkrétní třída se vybírá dle volby uživatele. Pro správu je použita implementace návrhového vzoru Factory [Kraval2002]. Jedná se o třídu *Interpolation*, která při zavolání metody **calculateColorInterpolation(…)** vybere dle nastavení příslušnou třídu pro výpočet interpolace, předá ji vstupní data a výsledek vrátí. Nejedná se tedy o čistou implementaci vzoru Factory, je spíše převzata myšlenka dynamické volby konkrétní třídy až dle aktuálního požadavku za běhu programu. Výsledkem je již výsledná barva bodu, kterou uložíme do objektu starajícího se o uložení textury do paměti (viz. další kapitola).

2.6.Výpočet souřadnic textury

Vzhledem k tomu, že model je tvořen trojúhelníky a textury lze ukládat pouze do čtvercového obrázku, je nutné si pamatovat souřadnice bodů trojúhelníku na textuře. Vzhledem k tomu, že je možno používat různé velikosti textury, byl obecně zaveden relativní souřadný systém pro ukládání souřadnic bodů na textuře. Pro souřadnice na ose X a Y se používají hodnoty od 0.0 do 1.0, přepočet na absolutní souřadnice (souřadnice pixelu) zajistí již grafický subsystém počítače. Bohužel zavedení počátku a směru souřadných systémů není univerzální pro všechny systémy. Situaci znázorňuje Obrázek 14. Systémy pro obrázky a rastrovou grafiku (typicky programovací jazyky, kreslící programy) používají jako počátek souřadného systému horní levý bod a osy směřují vpravo (osa X) a dolu (osa Y). Systémy pro 3D grafiku (například OpenGL) naopak používají jako počátek levý dolní roh, osa Y pak směřuje v opačném směru (tedy nahoru). Je tedy nutné si jeden ze systémů vybrat a při používání systémů druhého typu provést příslušný převod, což je přepočet souřadnice Y. V našem případě byl zvolen jako výchozí systém OpenGL, protože to ušetří přepočty při zobrazování pomocí vnitřního prohlížeče. Druhý souřadný systém je potřeba pouze při exportu do externích formátů, což je jednorázová operace, takže nám malé zdržení při přepočtu nevadí.



Obrázek 14 Souřadný systém textury Java (červeně) a OpenGL (zeleně)

Vlastní hodnoty se počítají dle postupu v teoretické části. Výstupem jsou dvojice hodnot typu double, které se ukládají k datům příslušné textury (viz. následující kapitola).

2.7.Uložení textury a souvisejících dat

Proces rekonstrukce je v tuto chvíli hotov, zbývá texturu vhodně uložit, aby bylo jednoduché ji znovu načíst, případně ji exportovat mimo náš program. Pro uložení textury slouží třída *TextureData*. Vlastní data o obrázku textury jsou uložena v instanci třídy *BufferedImage*, což je, dalo by se říci, základní třída pro reprezentaci obrázku v jazyku Java. Souřadnice textury jsou ukládány do mapy, kdy klíčem je ID bodu, a data jsou tvořena polem hodnot typu double. Z hlediska výpočtu by mohla tato data stačit, pro potřeby uložení dat na disk a pro zobrazení v interním OpenGL prohlížeči jsou ale potřeba další data.

Pro ukládání dat projektu je použit proces serializace implementovaný přímo v Javě, kdy je možno využít tříd *ObjectOutputStream* a *ObjectInputStream* pro velice jednoduché ukládání a načítání kompletních instancí tříd. Podmínkou je, že daná třída musí implementovat rozhraní <*Serializable*>, to ale nedefinuje žádné metody, pouze říká překladači, že tuto třídu lze ukládat pomocí výše uvedených objektů. Některé třídy Javy ale toto rozhraní neimplementují, je tedy nutné toto omezení nějak obejít. Je to i případ třídy *BufferedImage*, kdy je nutné data z této třídy převést na jinou reprezentaci a tu uložit. Samozřejmě při načítání se musí použít opačný postup. Pro naše potřeby byla zvolena reprezentace pomocí pole hodnot typu byte. Převod mezi třídou *BufferedImage* a polem je jednoduchý, existují pro to metody ve třídě *ImageIO*, převod je tedy pouze otázku přípravy pole a volání vhodné metody třídy *ImageIO*. Stejně tak zpětné načtení je pouze otázkou vytvoření vstupního proudu a zavolání příslušné metody třídy *ImageIO*.

Pro zobrazení textury pomocí OpenGL je nutné načíst data do paměti grafické karty. To znamená převést data ze třídy BufferedImage do reprezentace, které API grafické karty rozumí. Pro tuto operaci je k dispozici třída *TextureIO*, která umí převést mnoho reprezentací obrázku (jeden z nich je i BufferedImage) na instanci třídy *Texture*, kterou je již možno přímo načíst do paměti grafické karty. Informace o textuře na grafické kartě není nutné ukládat na disk, protože se musí znovu vytvářet při změně grafického kontextu (typicky nové spuštění aplikace), proto je položka „_texture“ ve třídě *TextureData* označena jako „transient,“ což říká překladači, aby tuto položku nezahrnoval do výstupního proudu dat při serializaci (uložení na disk).

2.8. Systém ukládání dat

Původní systém ukládání nebyl řešen zcela optimálně. Každá komponenta si řešila ukládání zcela po svém, neexistovalo nějaké společné rozhraní pro potřeby ukládání či načítání. S tím souvisí také neexistence nějaké třídy, která by zaštiťovala proces ukládání / načítání, čímž by bylo možno oddělit tento proces od zbytku programu. I když se toto oddělení může zdát zbytečně náročné, při vhodném návrhu se jedná o jednoduchou strukturu, navíc jakákoliv následná modifikace systému ukládání bude záležitostí několika málo oprav kódu, přičemž ve stávajícím systému ukládání je jakákoliv modifikace velice náročná.

Základní myšlenkou nového systému je jedna třída, která bude zpřístupňovat rozhraní pro uložení / načtení dat. Jak to bude vnitřně realizovat, to ostatní nezajímá, ostatním stačí vědět, že když zavolají například metodu **saveItems()**, tak dojde k uložení celého projektu. Tato zastřešující třída se jmenuje *PeristenceHandler* a lze ji nalézt v balíčku „cz.tul.data.utils.“ Veřejně dostupné jsou celkem tři metody – uložení dat, jejich načtení a test, jestli je ukládání potřeba. Důležité je také zmínit, že toto ukládání nevytváří výsledný soubor projektu, jedná se pouze o uložení do přechodné složky projektu. Zabalení do souboru projektu se realizuje až na pokyn uživatele a vlastní zabalení řeší jiná třída. Ukládání pomocí *PersistenceHandleru* je cíleno hlavně na automatické ukládání, samozřejmě je ale také využíván i při ukládání, které inicioval uživatel.

Další klíčovou myšlenkou je, že obalová třída nebude hledat třídy, které má ukládat, naopak samotné třídy se zaregistrují pro ukládání. Pro umožnění tohoto chování je nutné splnit dva předpoklady – dát k dispozici metodu pro registraci a definovat obecné rozhraní pro ukládání / načítání, které musí třída, která se chce registrovat, implementovat. Metoda pro registraci se jmenuje **addItem(…)**. Registrované položky jsou pak ukládány do seznamu, pomocí kterého se k nim pak přistupuje. Rozhraní pro ukládání je definováno pomocí abstraktní třídy *Saveable*. Čisté rozhraní nebylo zvoleno, protože metody pro správu nutnosti ukládání jsou společné pro všechny třídy implementující toto rozhraní, můžeme je tedy implementovat již na úrovni rodičovské třídy. Implementace jako třída nám také umožňuje využít třídu *Observable* (podědíme její vlastnosti), což nám zjednoduší sledování změn v datech jednotlivých tříd pomocí třídy *PersistenceHandler*. Poslední výhodou je možnost definice společné části

ukládání, což je kontrola existence složky, kde by měla být data a případná náprava (vytvoření složky v případě ukládání, vyhození výjimky v případě načítání).

Pokud tedy budeme chtít, aby se nějaká třída účastnila společného ukládání dat, musíme ji definovat jako potomka třídy Saveable, definovat metody **save()** a **load()** (můžeme využít již vytvořené rodičovské části pro kontrolu složky) a zaregistrovat tuto třídu do procesu ukládání ve třídě *PersistenceHandler*. O ostatní se již nemusíme starat, v případě nutnosti uložení se PersistenceHandler postará u nastavení složky pro ukládání / načítání a zavolání příslušné metody.

2.9. Export texturovaného modelu do VRML

V původním programu byl k dispozici export výsledného modelu do formátu VRML (byly k dispozici i další formáty). Formát VRML podporuje i zobrazení modelu s texturami, bylo tedy přistoupeno k rozšíření exportu o textury. Pouhý drátěný model lze zahrnout do jednoho tvaru (ve VRML se nazývá „Shape“), který bude tvořen jednotlivými polygony. Bohužel každý tvar si může držet pouze jednu texturu, texturovaný model musíme tedy vytvořit pomocí několika tvarů. Na vzhled výsledného modelu to ale nemá vliv (pomineme-li, že model bude texturován).

Textury je nutno k modelu dodat externě pomocí parametru „url,“ který obsahuje název a cestu k textuře. Je tedy nutné připravit export textur. K tomu jsou v jazyku Java metody třídy *ImageIO*. Umožňují uložit instanci třídy *BufferedImage* do daného souboru v daném formátu. Jedinou komplikací je nutnost jednoznačné identifikace souboru s texturou na základě jejího názvu. Název je tvořen indexy bodů polygonu, takže například polygon tvořen body s indexy 11,2,3 má stejný název jako 1,12,3. Proto jsou jednotlivé indexy odděleny v názvu souboru tečkou.

Do souboru VRML je nutno také uložit souřadnice textury, ty získáme z objektu textury. Pouze je nutné invertovat souřadnici Y, protože systém VRML používá jiný souřadný systém.

3. Testování

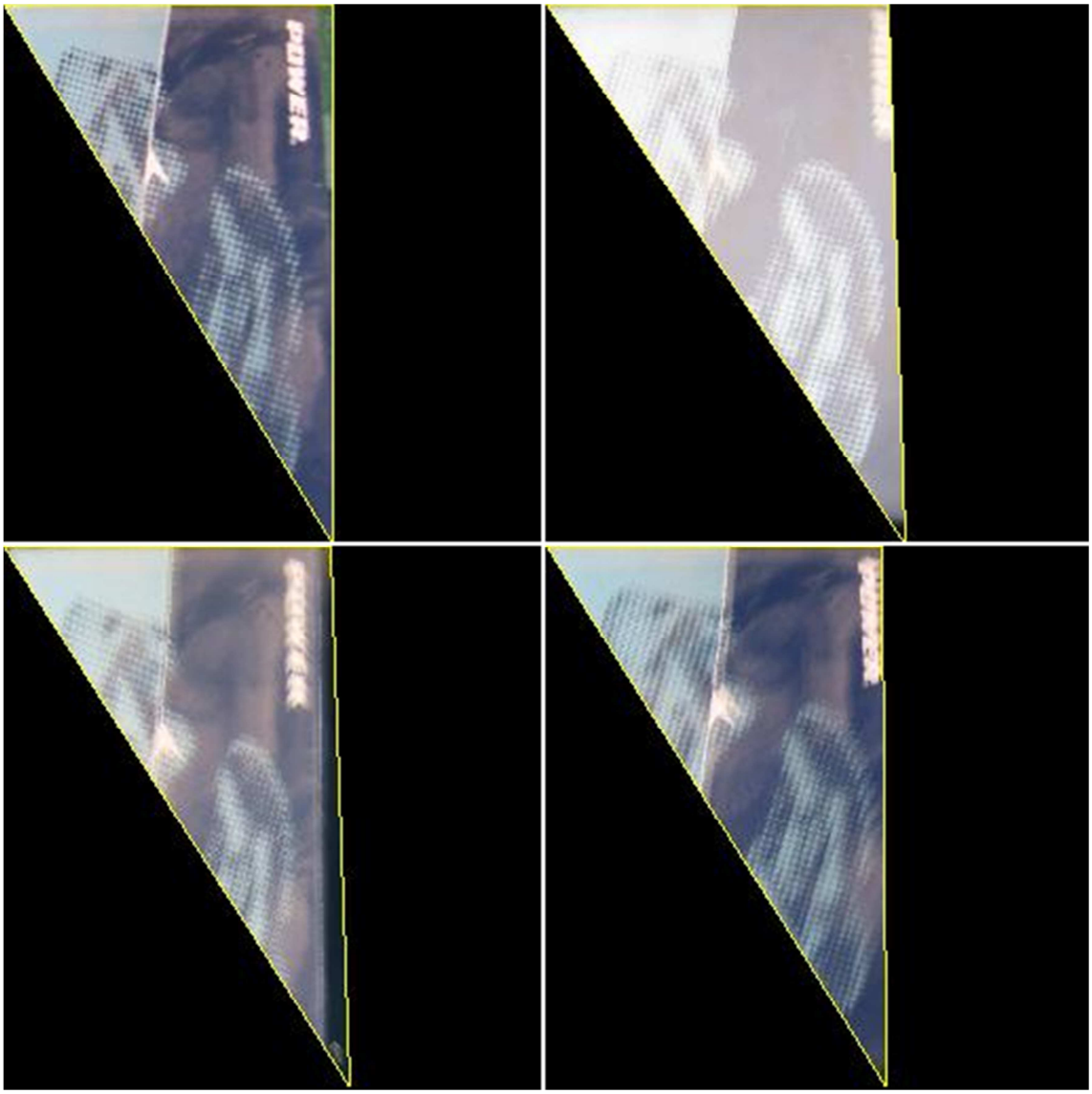
Algoritmus rekonstrukce máme nyní plně popsán. Teoretický postup ale není plným ukazatelem kvality vytvořeného algoritmu. Je nutné ho odzkoušet, jednak z hlediska vlastní funkčnosti (jestli vůbec algoritmus funguje) a samozřejmě z hlediska uživatelské přívětivosti. Uživatelskou přívětivostí se myslí například náročnost nastavení před vlastním výpočtem, protože nutnost složitého nastavování může odradit mnoho potencionálních uživatelů. Mnohdy může vést ke zmatení uživatele a možné degeneraci výsledků vlivem nevhodného nastavení. Dalším faktorem je také doba výpočtu. Čím déle bude výpočet trvat, tím spíše se uživatel poohlédne po jiném, rychlejším řešení.

Náš algoritmus vyžaduje pouze dvě nastavení – velikost výstupní textury a použitá interpolační metoda. Velikost textury se vybírá z předem daných hodnot, platí, že čím větší textura, tím kvalitnější výstup. Výběr interpolační metody může být pro neznalého uživatele nepříliš informativní, výchozí volba ale zajistí dostatečně kvalitní výsledky, takže se o tuto volbu uživatel nemusí nutně starat.

Testování se soustředí na dva cíle – vliv velikosti výstupní textury na čas potřebný k provedení výpočtu a vliv volby nejlepší fotografie na kvalitu textury. Pro testování byl použit jednoduchý objekt (krabice od bot). Díky menšímu počtu bodů lze totiž vcelku dobře eliminovat vliv chyb uživatele (hlavně nepřesné vyznačení polohy bodů na fotografii). Program byl samozřejmě odzkoušen i pro složitější modely, naším hlavním cílem je ale porovnání vlivu nastavených parametrů, takže si vystačíme s jednodušším modelem.

3.1.Volba nejlepší fotografie

Nevhodná volba fotografie může degradovat kvalitu výstupní textury natolik, že nemusí být vůbec použitelná. Nejhorší případy, kdy je textura zakryta, jsou z velké části eliminovány při vlastním výběru. Zakrytí ale není jediný faktor, který ovlivňuje kvalitu výstupu. Může to být také natočení plochy trojúhelníku vůči kameře, což by měl kompenzovat použitý matematický aparát. To se ukázalo jako pravdivé tvrzení. Obrázek 15 ukazuje výstupní textury pro jeden trojúhelník při použití různých fotografií.



Obrázek 15 Vliv volby nejlepší fotografie

Je patrné, že okraje trojúhelníku nejsou přesné, například vlevo nahoře je vidět kus zeleného podkladu. To je způsobeno buď nepřesností výstupního modelu, či nekvalitou rekonstrukce polohy a orientace kamery. K obdobné nepřesnosti došlo na fotografii vlevo dole, kde je vidět i kus dalšího trojúhelníku. Na obrázku jsou vidět i další deformace, které jsou často ale velice subjektivní a jejich automatická oprava bývá v některých případech velice problematická. Vpravo nahoře je patrné přesvětlení textury. To lze opravit pomocí algoritmů na vyvážení barev. Bohužel nezbyl čas na jejich implementaci a otestování, jedná se tedy o jedno z možných navázání na tuto práci. Další vadou je rozmazání fotografií (kromě levé horní prakticky všech), což bylo nejspíše způsobeno kombinací nevhodného nasvětlení scény a nastavení fotoaparátu. Oprava rozmazání je úloha obecně velice obtížně řešitelná, pokud dopředu nevíme co se má na obrázku nacházet a jaké má cílový objekt charakteristiky (například jestli se na něm nachází hodně hran, jestli jsou textury spíše pestré nebo jednolité apod.). Podobně jako u osvětlení nedochází k žádné opravě. Pokud by byl výstup nekvalitní z těchto důvodů, má uživatel možnost ručně vybrat jinou fotografii.

3.2. Závislost délky výpočtu na velikosti textury

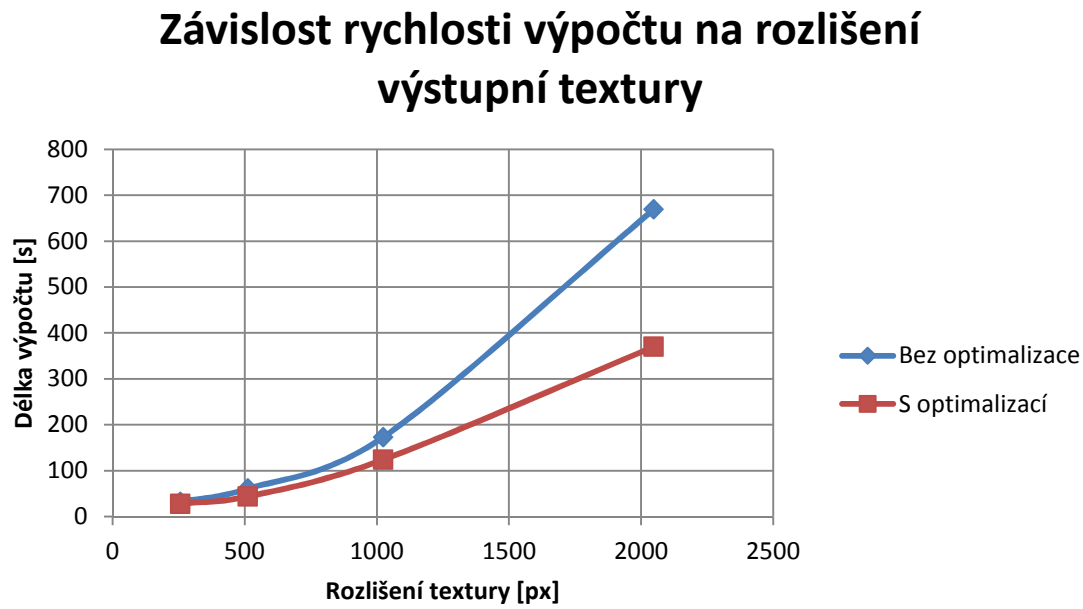
Uživateli samozřejmě nejvíce záleží na přesnosti výsledného modelu. Nemusí se nutně jednat o matematickou přesnost, spíše je důležité jestli objekt působí „přirozeně“ – strany jsou ve správném poměru, textury jsou správně namapovány a neobsahují nesmysly apod. Dalším požadavkem je ale také rychlost (či spíše délka) výpočtu. Pokud uživatel musí čekat mnoho hodin na dokončení výpočtu, nejedná se o příliš vhodný software. Proto se při výpočtech přistupuje k mnoha optimalizacím, které výpočet urychlí a přitom nijak nedegradují kvalitu výstupního modelu.

My se v této části soustředíme na jednu optimalizaci a jedno nastavení, které mají největší vliv na rychlost výpočtu. Optimalizací se myslí vynechání rekonstrukce bodů, které nejsou použity při zobrazování výstupní textury, nastavením pak je velikost textury. Bude tedy vyzkoušena rychlost výpočtu pro dva různé iterátory pro všechny dostupné velikosti výstupní textury. Ostatní nastavení výpočtu budou stále stejná. Bude použit iterační způsob nalezení nejoptimálnější polohy a orientace kamer při hustotě sítě 512. Jako interpolační metoda byla zvolena metoda nejbližšího souseda. Tímto získáme celkem deset variant výpočtu. Pro každou variantu bylo měření opakováno pětkrát, aby se alespoň částečně odstranil vliv úloh běžících na pozadí na stabilitu rychlosti výpočtu. Konečná hodnota rychlosti výpočtu pro každou variantu byla pak zvolena jako medián naměřených hodnot. Výsledky shrnuje Tabulka 1.

Velikost textury [px]	256	512	1024	2048
Bez optimalizace	32s	1m 1s	2m 53s	11m 9s
S optimalizací	28s	44s	2m 4s	6m 10s

Tabulka 1 Rychlost výpočtu v závislosti na nastavení

Z tabulky je patrné, že i pro nižší rozlišení znamená optimalizace vcelku postřehnutelnou úsporu času. Rozdíl se se zvyšujícím se rozlišením zvyšuje, pro maximální rozlišení textury 2048 pixelů je to rozdíl již velmi značný. Dalo se očekávat, že se zvyšujícím se rozlišením se bude rozdíl mezi optimalizovaným a neoptimalizovaným postupem zvětšovat, nebylo ale zcela zřejmé o kolik. Pro lepší představu byla data zanesena do následujícího grafu.



I přes celkem nízký počet bodů tvořících jednotlivé křivky je vidět, že se jedná o exponenciály. Není příliš důležité hledat přesný tvar funkce popisující naše křivky, hlavní je, že pro větší velikosti textury čas potřebný pro výpočet roste velmi rychle a optimalizace tento rychlý vzestup zpomaluje. Proto bylo rozhodnuto, že výpočet bude probíhat vždy za použití optimalizovaného výpočtu, protože nijak nedegraduje výsledek a volba by uživatele zbytečně mátla.

Závěr

Algoritmus rekonstrukce textury se podařilo realizovat ve velice dobré formě. Vlastní „narovnání“ textury je dostatečně univerzální, takže si poradí i s krajními polohami fotoaparátu (je myšleno natočení fotoaparátu vůči ploše daného trojúhelníku). Uživatel tedy není nijak omezován při hledání poloh, ze kterých bude cílový předmět fotografovat (samozřejmě musí dodržet omezení daná původním programem E3Dm). Výpočet přímo navazuje na rekonstrukci prostorového modelu. Z pohledu uživatele se tedy obsluha programu prakticky nemění (pouze může vybrat rozlišení textury a použitou metodu interpolace). Zadání diplomové práci bylo tedy splněno v plném rozsahu, navíc byl připraven prostor pro další rozšíření stávajícího programu.

Rychlost výpočtu je také velice dobrá, pro někoho by se mohlo zdát 6 minut jako dlouhá doba, musíme si ale uvědomit, že se jedná o maximální velikost textury, která je v dnešní době doporučována a je zaručena její globální podpora. Navíc se jedná o plochu jednoho trojúhelníku, nikoliv celého modelu, takže celkové rozlišení textury modelu může být mnohem větší. V krajních případech může uživatel přistoupit k vyznačení „falešných“ vazebních bodů, pomocí kterých navýší počet trojúhelníků a tím pádem i celkové rozlišení textury modelu.

Algoritmus vytvořený v této práci ale tvoří pouze kostru celého aparátu. Výpočetní jádro je hotové, možnosti pre- a post-processingu nebyly ale zdaleka využity. Preprocessing by se mohl rozšířit o chytřejší algoritmy hodnotící vhodnost fotografie. Máme k dispozici 3D model objektu, bylo by tedy možné například realizovat jakousi obdobu hloubkového testování z pohledu kamery, kdy bychom mohli jednoznačně říci, jestli je objekt zakryt i v případě, že se přímky pouze protínají. Dále by bylo možno hodnotit světelné podmínky, jestli se nejedná o příliš přesvětlenou fotografii či jestli není příliš rozmazána. Postprocessing by se pak zabýval opravou chyb detekovaných při preprocessingu. Zakrytí by se opravovalo velmi složitě, jednalo by se spíše o doostření fotografie, případně o vyvážení barev textury.

Seznam ilustrací

Seznam použité literatury

[Ječmen2009] JEČMEN, P. *Extrakce 3D grafického modelu z 2D dat*. Bakalářská práce. Liberec: TUL, 2009.

[Ječmen2010] JEČMEN, P. *Port aplikace „Extrakce 3D grafického modelu z 2D dat“ do jazyku Java*. Magisterský projekt, Liberec : TUL, 2010.

[Kraval2002]KRAVAL, I. *Design Patterns v OOP*. 1. Vydání, http://www.objects.cz, 2002.

[Přívratská2007] PŘÍVRATSKÁ, J. *Geometrie* pro techniky. *Modul 1*. Liberec : Technická univerzita v Liberci, 2007, ISBN 978-80-7372-194-7

[Švrček1988] ŠVRČEK, J. *Geometrie trojúhelníka*. 1. Vydání, Praha : SNTL, 1988, ISBN 80-7184-584-1.